REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YY)* 11-20-10	**2. REPORT TYPE** Final Report	**3. DATES COVERED *(From - To)*** 01-12-10 to 11-27-10

4. TITLE AND SUBTITLE Safety In Numbers	**5a. CONTRACT NUMBER** N00014-10-C-0225
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) David Cok	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) GrammaTech, Inc. 317 N. Aurora Street Ithaca, NY 14850	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 875 North Randolph St. Suite 1102B Attn: Sukarno Mertoguno Arlington, VA 22203-1995	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** SMDC-RDT
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution Statement A: Approved for public release; distribution is unlimited.

DESTRUCTION NOTICE – For classified documents, follow the procedures in DOD 5220.22-M, National Industrial Security Program Operating Manual (NISPOM), Chapter 5, Section 7, or DOD 5200.1-R, Information Security Program Regulation, Chapter IX. For unclassified, limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Using large-scale distributed resources can help find vulnerabilities and malicious code. This project studied the feasibility of distributing two kinds of static analyses of machine code across large-scale donated computational cycles: conventional static analyses for finding bugs and vulnerabilities, and concolic execution to find test cases that trigger rare, possibly maliciously hidden, code paths. We demonstrated that concolic execution is particularly suited to large-scale distributed execution since its core computational loop is very parallelizable and communication costs are small. We assessed a large number of possible parallel architectures and experimented in depth with three. In the process of expanding and scaling our concolic engine for this application, we also devised a means to ‘fuzz’ its semantic representation of machine code and so were able to demonstrate a general technique for validating abstract representations of machine code semantics.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**		104	Sukarno Mertoguno **19b. TELEPHONE NUMBER *(Include Area Code)*** 202.577.3887

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Project Final Report
Report Date: November 27, 2010
For the period 1/12/2010 to 11/27/2010

Contract Number: N00014-10-C-0225
Sponsor: Office of Naval Research (ONR)

Title: Safety In Numbers

Submitted by: GrammaTech, Inc.
317 North Aurora Street
Ithaca, NY 14850-4201

Project Manager:	**Financial Data Contact:**	**Administrative Contact:**
David Melski	Krisztina Nagy	Krisztina Nagy
Tim Teitelbaum	(607) 273-7340 x17	(607) 273-7340 x17
(607) 273-7340 x23	(607) 273-8752 [fax]	(607) 273-8752 [fax]
(607) 273-8752 [fax]	knagy@grammatech.com	knagy@grammatech.com
melski@grammatech.com		

Contributors: David Cok, John Phillips, Scott Wisniewski, Suan Hsi Yong, Nathan Lloyd, Lindsay Kuper, Denis Gopan, Alexey Loginov

Abstract

Using large-scale distributed resources can help find vulnerabilities and malicious code. This project studied the feasibility of distributing two kinds of static analyses of machine code across large-scale donated computational cycles: conventional static analyses for finding bugs and vulnerabilities, and concolic execution to find test cases that trigger rare, possibly maliciously hidden, code paths. We demonstrated that concolic execution is particularly suited to large-scale distributed execution since its core computational loop is very parallelizable and communication costs are small. We assessed a large number of possible parallel architectures and experimented in depth with three. In the process of expanding and scaling our concolic engine for this application, we also devised a means to 'fuzz' its semantic representation of machine code and so were able to demonstrate a general technique for validating abstract representations of machine code semantics.

Table of Contents

List of Figures

s

Section I: Project Overview

1 Summary

Finding malicious or vulnerable code in masses of software is a challenging and computationally-intensive task. Highly-distributed computing using donated resources has been used to good effect on challenging, computationally-demanding scientific problems. This project investigated applying parallel or distributed computation at large scale to finding more code vulnerabilities than can simple time-bound serial execution. The results of the project are summarized under the following five headings:

1) [Section **Error! Reference source not found.**] We assessed the advantages and disadvantages of using highly distributed computing with volunteered computational resources. The advantages are straightforward: more computation is performed. The disadvantages are primarily social and security-related: can the various participants in a donated-computation relationship trust each other. The person donating computational cycles must trust that the computation being performed will not overly tax the donated computer and will not act maliciously or carelessly; the person doing the computation must trust that the donated resources will neither subvert nor expropriate the computation being performed.

The actual technical work of distributing a computation across a volunteer network has been well-demonstrated and we did not repeat that work here. What is needed is to determine whether the desired computation can be divided into sufficiently independent computational units and can be performed efficiently in a parallel fashion. We investigated that question in Sections 6 and 7.

2) [Section **Error! Reference source not found.**] The project investigated whether applying more computation to conventional static analysis would increase the number of problems found. In particular, we studied a GrammaTech tool that finds problems in binary code (without source code) – the CodeSonar/x86 tool. The tool has a number of internal parameters that are heuristically set to constrain the time consumed by the computation to reasonable bounds. By relaxing those restrictions, we investigated what improved bug-finding performance we could obtain with additional processing. We found that, while some additional benefit was obtained, it was not worth the additional time taken, and not worth the considerable effort to parallelize the computation. This is not to say that the static analysis cannot be improved; rather that significant improvements will require changes to the underlying logic and reasoning engine, rather than by adjusting parameters of the existing algorithms.

3) [Section **Error! Reference source not found.**] We also investigated a second technique for finding problems in machine code: concolic execution. In particular we investigated parallelizing GrammaTech's GraCE tool. This tool executes a subject program on a given input, records the machine code instructions that are executed, translates the execution (including all the branches taken) into logical constraints, and then uses a constraint solver to find alternate program inputs that will cause the program to take different branch directions and therefore new program paths.

The tool generates a high-coverage test suite that exercises many combinations of branches and many different execution paths.

This tool is highly parallelizable. We assessed many alternative parallel architectures for the prototype, choosing three for implementation and experimentation. One simple architecture is to simply start N copies of the analysis engine on different random inputs, with no communication other than assembling all of the results at the end. We found that for typical programs the independent processes were prone to repeat work that other processes had already performed; hence the overall efficiency of using multiple processors suffered significantly.

An improvement is to carefully choose the starting inputs for the N independent processes to avoid some of the repeat work. This does improve the overall efficiency, but not always significantly, and the technique does require expert knowledge of the subject program in order to choose good inputs.

The best architecture is a shared queue for program inputs to be analyzed, managed by a single controller process; identical worker processes take inputs from the queue, analyze the inputs, and adding new analysis tasks back to the queue; as each worker completes its analysis task, it obtains a new task from the shared queue. The communication overhead (of the program inputs and the analysis results) is very modest.

This architecture is very suited to a network of volunteer processors; it can readily adapt to a varying supply of available computation and to processors that may be unreliable or even to some degree untrustworthy.

4) [Sections 8, 9] We made reliability and scalability improvements in the overall GraCE tool in order to carry out the experiments described above. The most interesting of these was turned into a research investigation of its own, described in the next paragraph.

5) [Section 10] A tool that reasons about machine code programs implicitly encodes a translation of each machine code instruction into a logical statement. In our tool we make that translation explicit through a language called TSL [59]. TSL provides a framework in which each machine code instruction is mapped to a corresponding statement in a Quantifier-Free Bit Vector (QFBV) logic. For any such translation, one can ask this question: is the representation as a logical constraint correct? If it is not, the reasoning about the machine code will not match the actual execution of the machine code.

We decided to test the QFBV representation as follows. We automatically generated small programs consisting of single machine code instructions. We prepared a random initial state of memory and registers and executed the instruction on that state. We also abstracted a logical representation of the initial state, the instruction semantics, and the final state. For each test we can compare the actual final state with the logical final state obtained by applying the logical abstraction of the instruction to the abstract initial state. The results of many such tests over the large number of x86 instructions and many instantiations of random initial states are presented in the full report; there was not time within the contract to understand all of the discrepancies –

some are outright bugs in the semantics and others represent undocumented subtleties of the machine instructions.

We also demonstrated a proof of concept prototype that the technique of the previous paragraph can be applied to a variety of abstract interpretations of underlying machine semantics, and not just to a specific concrete interpretation of the machine code.

Conclusion: The overall conclusion of the project is that highly distributed processing can be used to advantage in checking software for bugs or vulnerabilities. In particular, deriving test suites through concolic execution is particularly amenable to parallel processing. The project demonstrated this through experiments with a prototype on open source software obtained from public sources.

2 Overview

This report summarizes the technical accomplishments of the Safety in Numbers project, sponsored by ONR and executed by GrammaTech. Section 3 provides some programmatic information. Section 4 describes the motivation and background for the project. Section 5 summarizes the areas of technical effort; detailed reports on each of these are then provided in Sections 6-11, each describing its own methods and conclusions. Section 12 describes previous work in these areas, and section 13 summarizes overall conclusions and recommendations.

3 Programmatics

3.1 Financial Summary

Total contract amount (one year)	$300,000.00
Costs incurred during the performance period	$300.000.00

3.2 Milestone/Deliverable Schedule

Milestone or deliverable	Planned Delivery/ Completion Date	Actual Delivery/ Completion Date
Create work environment	02/28/2010	02/25/2010
Complete architecture review	02/28/2010	02/28/2010
Identify parametric constraints in the static analysis engine to relax in the parameter study	02/28/2010	02/20/2010
Develop plans for different ways to parallelize a concolic execution engine	03/30/2010	03/25/2010
Submit April quarterly report	04/12/2010	04/12/2010
Measure potential benefits of coarse grained	04/12/2010	03/20/2010

parallelism for concolic execution engine		
Analyze measurements on concolic engine	04/20/2010	03/24/2010
Select a plan for the first prototype	04/21/2010	03/25/2010
Develop detailed design for first prototype	04/25/2010	03/30/2010
Begin coding of first prototype	04/25/2010	03/30/2010
Measurements of the effect of changing parametric constraints on the static analysis engine	04/30/2010	03/29/2010
Determine list of prototypes to develop	04/30/2010	4/30/2010
Analyze measurements on static analysis engine	05/30/2010	7/14/2010
Complete first working prototype	05/31/2010	7/7/2010
Decision on benefits and plan for parallelization of static analysis engine	06/15/2010	06/15/2010
Build testing infrastructure for correctness, robustness, performance, and coverage	07/01/2010	06/25/2010
Identify issues for scalability and robustness of GraCE	06/30/2010	06/20/2010
Modify GraCE to address identified issues	10/01/2010	10/15/2010
Validate representation of machine code semantics used by GraCE	09/01/2010	10/31/2010
Submit July quarterly report	07/12/2010	07/13/2010
Assess the benefits and pitfalls of using large scale parallel computation to detect malware	07/15/2010	8/31/2010
Complete report on using large scale parallelism to detect malware	07/31/2010	11/27/2010
Complete implementation of all planned prototypes	07/31/2010	8/15/2010
Submit October quarterly report	10/12/2010	10/12/2010
Complete assessment of prototype performance on large application benchmarks	11/01/2010	11/15/2010
Submit final report on first year's work	11/27/2010	11/27/2010

3.3 Personnel

The following personnel contributed to the project activities.

Prof. Tim Teitelbaum was a co-PI for this effort. Teitelbaum is Chairman of GrammaTech and is also a faculty member in the Department of Computer Science at Cornell University, which he joined in 1973. His pioneering work on programming environments and incremental computation is widely cited in the literature. His Publications include [9, 16, 30, 69, 75].

Dr. David Melski was co-PI of the project. He is the vice-president of research at GrammaTech and has overseen many successful projects. Dr. Melski graduated *summa cum laude* from the University of Wisconsin in 1994 with a B.S. in Computer Sciences and Russian Studies. He received his Ph.D. in Computer Sciences from the University of Wisconsin in 2002, where his research interests included static analysis, profiling, and profile-directed optimization. His publications include [17, 46, 64, 70].

Dr. David Cok, hired into GrammaTech in January, collaborated in directing the scientific direction and implementation work of the project. He is a technical expert and leader in static analysis, software development, computational science and digital imaging. He has been a major contributor to JML and Esc/Java2 for Java applications; his particular interests are the usability and application of static analysis and verification technology for industrial-scale software development. Prior to GrammaTech, Dr. Cok was a Research Fellow in the Kodak Research Laboratory and held technical leadership and managerial positions. He received a Ph.D. in Physics from Harvard University in 1980.

Dr. Denis Gopan is an expert on concolic processing and will contribute to the analysis and rearchitecting of the concolic engine. He received a B.Sc. in Computer Science from the University of Wisconsin-Milwaukee in 1996 and a Ph.D. in Computer Science from the University of Wisconsin-Madison in 2007. His research interests are static program analysis and verification. While at Wisconsin, Dr. Gopan was awarded the CISCO fellowship for two consecutive years.

Mr. John Phillips has many years of experience in parallel computing, high performance computing, and C++, including more than nine years of involvement in the Boost collaboration. Prior to joining GrammaTech in January 2010, he was a member of the faculty at Capital University in Mathematics, Computer Science, and Physics and completed several NSF and DoE-funded projects in Astrophysics and Computational Science.

Other contributing engineers were **Alexey Loginov, Scott Wisniewski**, **Ducson Nyugen**, and **Suan Hsi Yong,** and summer interns **Lindsey Kuper** and **Nathan Lloyd.**

4 Introduction and Motivation for the project

The goal of the Safety In Numbers project is to find improved ways to detect malicious code in otherwise commonly used software and, in particular, to do so by leveraging possibilities for parallel static and hybrid analysis using multi-core processors or multiple processors in local or large-scale grid combinations.

The global information grid has changed the nature of what is possible in a very short time span. One important effect is the new ways in which communities form and gain leverage on previously intractable problems. For example, it is now possible for an encyclopedia written by volunteers (Wikipedia) to rival those that are collated by professionals. Similarly, Linux has demonstrated that operating systems are no longer solely the domain of well-funded, commercial enterprises. Google's online email system, Gmail, leverages feedback from its user community to create effective spam filters. Programs such as Folding@Home rely on a community willing to donate spare computational cycles to help with research on causes of disease and the development of new treatments. On the darker side, malicious actors are able to co-opt tens of thousands of user machines to create artificial communities (botnets) that launch cyber attacks or send spam.

None of these are new observations. Many books have been written about the significance of the digital age, and we cannot hope to summarize all of its ramifications here. Our concern is with developing a new approach for improving the security of computer systems, one that leverages the new realities of the global information grid, rather than fighting them. As mentioned above, online communities can provide new leverage on intractable problems. In this project, we focused on the problem of detecting malicious code, although we anticipate that the work will also be applicable to defect detection. (Unintentional defects and malicious code can share many characteristics and may be hard to distinguish.) The 2007 report from the DoD Defense Science Board Task Force on the "Mission Impact of Foreign Influence on DoD Software" identified detection of malicious code as a high priority for the DoD [2]:

> *There are two distinct kinds of vulnerabilities in software. The first is the common "bug", an unintentional defect or weakness in the code that opens the door to opportunistic exploitation. The Department of Defense shares these vulnerabilities with all users. However, certain users are "high value targets", such as the financial sector and the Department of Defense. These high-value targets attract "high-end" attackers. Moreover, the DoD also may be presumed to attract the most skilled and best financed attackers—a nation-state adversary or its proxy. These high-end attackers will not be content to exploit opportunistic vulnerabilities, which might be fixed and therefore unavailable at a critical juncture. Furthermore, they may seek to implant vulnerability for later exploitation. It is bad enough that this can be done remotely in the inter-networked world, but worse when the malefactors are in DoD's supply chain and are loyal to and working for an adversary nation-state—especially a nation-state that is producing the software that the U.S. Government needs.*

There is no simple approach to eliminate the threat of malicious code. Building trustworthy software is a cradle to grave operation, and may require dramatic changes in current practices.

Trust issues must be considered during design, development, integration, deployment, daily use, maintenance, and upgrading. Accounting for who authored a piece of software and under what conditions may be as useful as examining code when judging malicious intent.

At the end of the day, answering questions about software malice must include analysis of the machine code that will actually run on the hardware. This is partly because of the *WYSINWYX effect*. WYSINWYX stands for "**W**hat **Y**ou **S**ee (in source code) **I**s **N**ot **W**hat **Y**ou e**X**ecute (on the machine)." The WYSINWYX effect can arise for many reasons including source-language ambiguities, compiler errors, and use of third-party components. The significance of the WYSINWYX effect is that it can undermine any evaluation of software that is based (solely) on artifacts generated during the software's development, up to and including the source code. As a practical matter, source code is often unavailable, to say nothing of other data concerning the software's development. For these reasons, our primary interest will be in techniques that make analysis of machine code more effective. The findings of the Defense Science Board recognize the need for machine-code analysis and explicitly cite GrammaTech's work as one of three research efforts on machine-code analysis that "are already beginning to bear substantial fruit" [2].

Machine-code analysis presents a challenging technical hurdle for many reasons. Few people are trained in understanding machine code. Machine code comes without helpful information found in source code (such as types). There have been many recent breakthroughs, including *value-set analysis* [13], *concolic execution* [47, 72], and *verification by testing* [20]. However, there are still major challenges in achieving scalability and performing analysis of entire systems. We believe that online communities provide new leverage on these problems in many ways, including:

1. *Donation of spare computation cycles.* In this model, community members volunteer to run partial computations on their (personal) computers when they are otherwise idle. The partial computations are then assembled into a complete problem solution. In this case we refer to the community as a *computational community* that performs a distributed computation. In the past, this approach has been used to demonstrate the danger of using short (56-bit) keys for encryption (e.g., with DES or RC5) [45]. Today, it is used to understand protein folding [10]. IBM's World Community Grid is slated to help discover new materials for use in solar cells [37]. We envisage that donated cycles might enable deeper and more accurate analysis of machine code. In particular, this may be an instance where the quantitative increase in computation power leads to a qualitative change by enabling more expensive analyses and allowing them to be applied to entire software systems.

2. *Passive feedback from user communities.* Using this approach, community members allow observation of their computer systems under normal usage. The observations are used to build more accurate models of their software. The feedback is passive in the sense that community members do little or nothing to generate the feedback beyond their

normal interactions with the system. A simple but effective example of this approach is the Dr. Watson tool on Windows platforms. Dr. Watson allows Microsoft to collate reports about program crashes and has proven very useful for improving platform reliability [55]. Liblit developed a more sophisticated approach called *Cooperative Bug Isolation* that incorporates statistical analysis of the feedback in order to isolate bug causes [56]. In our case, we believe passive feedback could result in more accurate models of the system that are useful for identifying malicious behavior.

3. *Active feedback from user communities.* This technique leverages the wisdom of the community by soliciting active feedback from community members. As mentioned above, Gmail leverages user feedback (via the "Report Spam" button) to train its spam-filtering tool. Recommendation systems such as Netflix's "Cinematch" rely on user-provided ratings to identify similarities in taste and suggest movies to customers. One of the challenges we face is that "malice" can be context dependent. An application that broadcasts your current GPS coordinates may take off as a "cool new app" for the iPhone, but the same feature on a Navy system would be treasonous. In this sense, identifying malice may be similar to suggesting movie titles, but with much higher stakes. In general, it is also much more difficult for community members to reliably identify malicious software behavior than their own movie preferences. Nevertheless, Gmail has demonstrated that at least in some cases, user feedback is valuable for identifying malice (i.e., spam).

Leveraging community support (as outlined above) comes with its own set of technical challenges. The entire community effort is at risk if malicious actors can subvert or co-opt the results, for example, by feeding false information to coordinators. An open community built from volunteers may contain spies who could set up competing communities. For example, organized crime could steal a distributed security analysis, run the analysis on a botnet, and discover and exploit vulnerabilities before they can be patched. Feedback mechanisms (either passive or active) run the risk of leaking the private or classified information of the community members.

There are also challenges that are more economic or political than technical. These issues must not be ignored: too many great technical solutions to the problem of computer security have failed because they did not consider the non-technical requirements. For example, proof-carrying code is an approach for guaranteeing that software is free of certain vulnerabilities. Microsoft's Singularity project demonstrated a novel architecture for an operating system that is more secure (and possibly more efficient) than existing OS architectures. However, in important ways, these efforts are at odds with existing economic realities of software development and deployment. Proof-carrying code requires new development and deployment tools. Singularity was orphaned because it would be too costly to provide backwards compatibility with existing products. Java is another effort that attempted to tackle software security issues with a novel platform architecture (the Java VM). Arguably, it has been more successful than other efforts because Sun marketed it with an economic incentive for its adoption (i.e., cross-platform portability).

In this single year project, we have tackled one of the above research challenges: determining the value of parallel processing for finding vulnerabilities in software. The ability to distribute a sufficiently parallel computation across a network of volunteer computers has been demonstrated in actual practice; thus, in this project we concentrated not on the details of cloud computation or obtaining donated cycles, but rather on the question of whether static analysis for finding malicious code or simple vulnerabilities can actually be distributed across many computers to advantage.

GrammaTech was well-positioned to investigate this question. It has a staff of researchers with extensive experience in program analysis; it has a facility clearance and employees with secret and top-secret clearances; and it has a long-standing research collaboration with highly successful academic research in program analysis at the University of Wisconsin which informs our research even in cases where U. Wisconsin is not a direct contributor to the research work.

Section II: Technical Accomplishments

5 Summary of Technical Accomplishments

The technical work of this project can be grouped into the following topic areas. Reports on each of these topics are provided in sections 6 - 11.

1. Assessment of parallel processing options in the static analysis tools CodeSonar and CodeSonar/SWYX. [Section 6]
2. Assessment and optimization of the parallel processing options in GrammaTech's concolic execution engine (GraCE). [Section 7]
    a. Evaluating many designs for parallel processing
    b. Implementing three chosen designs
    c. Measuring the performance of the designs along several axes
3. Testing and improving the underlying technology [Sections 8 and 9]
4. Validating the instruction semantics [Section 10]
    a. Validating the concrete semantics underlying GraCE
    b. Implementing a method for testing abstract representations
5. Evaluating the issues with using the cloud for analyzing malicious code [Section 11]

Relationship to the Statement of Work

The deliverables of the contract are interim and final reports describing the research findings. The tasks of Statement of Work are listed below.

1. *The contractor will investigate techniques based on the donation of spare cycles by a grid of computing resources.*

    We investigated existing techniques for harvesting spare cycles. Our parallel implementation of the concolic engine works side-by-side with other activity on the machines being used, using their spare cycles. Our conclusions regarding this task are described in sections 7 and 11.

2. *The contractor will investigate existing techniques for grid computation and techniques to safely leverage grid computation for analysis of malicious code and software security.*

    We investigated two approaches: static analysis tools and the concolic engine. Section 6 contains the analysis of parallelizing static analysis and section 7 describes the work on the concolic engine.

3. *The contractor will identify one promising approach based on grid computation investigate it in depth.*

    The more promising approach was the concolic engine, GraCE. We investigated many designs, and then we implemented and measured the performance of three designs in local networks. The results enable us to make projections about extrapolations to larger-scale, grid-organized computation. These results are described in Section 7, with related work described in sections 8-10.

4. *The contractor will generate a report describing potential uses and pitfalls of applying grid computation to program analysis.*

   This report is contained in section 11.

5. *The contractor will brief the government on progress of the research upon request.*

   Quarterly reports were submitted in April, July, and October 2010. On on-site briefing occurred on July 12. Briefing material was supplied as requested in March and April 2010.

6. *The contractor will write status reports for the government upon request.*

   An introduction and current summary was supplied as requested in March 2010.

7. *The contractor will generate a final report summarizing the results of all research.*

   This report fulfills this item.

6 Parallel static analysis

The project assessed two technologies as candidates for aiding the finding of malicious code using distributed computation. One of those technologies is static analysis as implemented in GrammaTech's CodeSonar tool. (The other technique – concolic execution is described in section 7.) This section describes the experiments we conducted to determine what gains could be obtained by parallelizing static analysis.

6.1 Static analysis on machine code

Recent research in programming languages, software engineering, and computer security has led to new kinds of tools for analyzing code for bugs and security vulnerabilities [18, 24-26, 32, 38, 42, 49, 50, 78]. In these tools, static analysis is used to determine a conservative answer to the question "Can the program reach a bad state?" Static analysis provides a way to obtain information about the possible states that a program reaches during execution, but without actually running the program on specific inputs. Static-analysis techniques explore the program's behavior for all possible inputs and all possible states that the program can reach. To make this feasible, the program is "run in the aggregate"—that is, on descriptors that represent collections of memory configurations [33].

In principle, such tools would be of great help to an analyst trying to detect malicious code hidden in software, except for one important detail: the aforementioned tools all focus on analyzing *source code* written in a high-level language. GrammaTech and the University of Wisconsin collaborated on the problem of finding bugs and security vulnerabilities in programs when source code is unavailable. Our goal was to create a platform that carries out static analysis on executables and provides information that an analyst can use to understand the workings of potentially malicious code, such as COTS components, plug-ins, mobile code, and DLLs, as well as memory snapshots of worms and virus-infected code. A second goal is to use this platform to create tools that an analyst can employ to determine such information as

- whether a program contains inadvertent security vulnerabilities and
- whether a program contains deliberate security vulnerabilities, such as backdoors, time bombs, or logic bombs.

We have developed a tool, called CodeSurfer/x86, that serves as a prototype for a next-generation platform for analyzing executables. CodeSurfer/x86 provides a security analyst with a powerful and flexible platform for investigating the properties and possible behaviors of an x86 executable. It uses static analysis to recover an IR that is similar to those that a compiler creates for a program written in a high-level language [13, 54, 67, 68, 70].

Despite the difficulty in performing IR recovery, CodeSurfer/x86 has advanced far enough to start producing results. Balakrishnan and Reps recently demonstrated the use of IR recovery in analysis of Windows device drivers [15]. Specifically, [15] demonstrated two results. First, that CodeSurfer's IR recovery produces precise results on device drivers. Second, that by building on the recovered IR, the researchers could adapt a technique for analyzing the source code of device drivers [19] to analyze machine code and find some of the same driver errors—while also side-stepping the WYSINWYX effect [17] and garnering the benefits of machine-code analysis.

6.2 The question: are more bugs found with more computation?

In static analysis, the source or machine code of a software program is analyzed without actually executing it. The instructions of the program are converted into logical statements in an abstract representation of the program. Then automated reasoning engines are applied to that logical representation to determine if various illegal or undesirable program states can be reached during execution. If such states are or appear to be possible, the static analysis engine attempts to find specific examples of program variables that can cause the illegal or undesirable state. Such examples help the user decide whether the situation is indeed possible, given the context in which the program is used, and the importance of the problematic condition.

Static analysis on a large program is very computationally intensive. Thus the results of practical static analysis tools are not completely precise. Sometimes warnings of undesirable states are given that will not actually occur in practice (false positives). This can happen because the static analysis is only partial and cannot accurately assess the whole program without substantial user input. For ease of use, most bug-finding static analysis tools avoid requiring significant amounts of user input.

The parameters of a static analysis tool can be adjusted to avoid false positives by only reporting problems for which there is a clear path from the beginning of the program. However, then the tool will miss many other actual errors (false negatives). For example, it may take a more than tolerable amount of computing time to find a path to some actual error; if the path is not found, then an error that is actually present in the code will not be reported. False negatives and positives can also occur because the logical abstraction of the program is only an approximation of the true program.

In practice, a practical static analysis tool uses heuristically set limits on its logical reasoning and search for error paths so that the tool efficiently finds possible errors in a reasonable amount of time; this includes finding a useful balance between false positives and false negatives.

The question we asked in this study was whether increasing the limits on computation time and altering internal parameters accordingly would cause more useful warnings to be found. For example, suppose using twice the computation time would find twice the number of warnings; then it would be worth asking (a) do those warnings correspond to true positives (actual errors), at least as frequently as the warnings reported previously and (b) is there a parallel computation design for the program that could accomplish that work with two processors without increasing the overall clock time taken. However, if we do not find appreciably more warnings by increasing the time taken, then it is not worth investigating parallel designs for the static analysis tool.

6.3 Procedure

To investigate this question we ran CodeSonar's SmashProof engine for several thousand hours, varying a number of internal parameters that regulate the tradeoff between analysis depth and performance. We allowed the tool to investigate the subject programs with more time than it would under standard conditions and recorded the number of warnings reported in each case.

We did not assess whether the additional warnings were true or false positives; we assumed that the additional warnings would be true positives in the same proportion as the original warnings. There are arguments in either direction on this point. On the one hand, one might argue that "most bugs are shallow", so that most real bugs are found without undue computation; additional effort may indeed find more but less efficiently as more effort is expended. On the other hand, the additional computation may be better at differentiating the bugs that with little effort appear to be possible into those that, with more effort, are clearly not possible and those for which a valid but lengthy path can be found. In that case the extra effort will be more effective on all warnings. However, as will be apparent in the data presented below, we did not find a sufficient number of extra warnings to warrant determining which warnings were true and which false.

Also, we did not actually run the computation in a parallel fashion. To do so would have required a substantial programming design and implementation effort, with unclear future benefit. Instead, we ran serial processes for longer. By doing so, we measure an upper bound to the benefit achievable by parallelization. Typically the speed-up obtained by parallelization is less than linear, because of the need for communication among cooperating parallel processes and the inability to parallelize all of the computational steps.

6.4 The test plan

The testing plan included three different facets, described in the subsections below. Each facet addresses a class of possible large scale parallel designs. The experiments and results for the three pieces of the test plan are presented in detail in sections 6.5, 6.6, and 6.7.

6.4.1 Using non-determinism

The CodeSonar front end constructs an intermediate representation of the input (source code or binary), and passes the representation to the analysis engine. The engine is non-deterministic[1], so one possible parallel design is to leverage that non-determinism. If the results of different instances of the engine acting on the same representation are mostly different, then a set of independent agents working on the same code base would produce largely independent sets of warnings. This would be ideal for large scale parallelism since the only information shared among the agents would be the initial representation and the final results; this initial representation can be either communicated from a central server or generated on the spot

[1] Due to bounds on the number and types of paths explored, as well as time out bounds imposed on how long one part of an exploration can take. These bounds are needed because the unbounded version of the analysis is unsolvable. The engine also uses heuristics to guide the imposition of some bounds.

depending on the cost of communicating it; the final results would be transmitted to a central server and collected there.

6.4.2 Relaxing time constraints

A different potential design increases the communications cost slightly, but still allows for large computational passes between communications. The Smashproof analysis engine creates slices through the intermediate representation and analyzes the contents of those slices. If different instances of the engine work on different slices, then that segment of the analysis can be largely independent. Some communications are needed to coordinate which machine is working on what slice, but otherwise the communications needs are similar to the independent agents. In a reasonable amount of user time, this would allow the time out bounds to be relaxed, and the deeper exploration allowed by the relaxed time out bounds might greatly improve the quality of the analysis.

6.4.3 Adjusting engine characteristics

Finally, a segment of the testing was devoted to determining which characteristics of the engine most influence the quality of the results. This information can be used to spark ideas for designs not already considered.

6.5 Using non-determinism to fuel independent agents

In this study, analyses of 5 different open-source programs[2] with a total of about ¾ million lines of code were repeated 20 times each. For each individual program, the total number of unique warnings generated[3] was calculated. The order of the 20 runs is arbitrary; new warning fractions were calculated from a specific ordering by determining what fraction of the total number of unique warnings were first seen in each analysis run, if the runs are explored in the arbitrary order assigned to the runs.

In general, the results were consistent in that the first run produced approximately 90% of all unique warnings, while later runs produced no more than 2-3% additional unique warnings each.

The graph in Figure 1 shows a cumulative view of the results. The “Fraction of total” measurements are the median across the 5 test programs for the fraction of unique warnings revealed in each analysis run. We could imagine performing these analyses in parallel and combining the results at the end. Then, a traditional measure for how much benefit is derived

[2] The programs are open source packages with wide use and extensive testing: uucp with ~50,000 lines of code (KLOC), Berkley database with ~94 KLOC, bind with ~142 KLOC, binutils with ~198 KLOC and samba with ~264 KLOC.

[3] CodeSonar provides fingerprinting of warnings so the identity of a given warning is the same over multiple runs, and even when there have been changes in the code base that do not affect the warning. This makes identifying unique warnings easy.

from the parallelization is the parallel efficiency.[4] The equivalent parallel efficiency is calculated using the assumption that the communications costs of sharing the representation and collecting

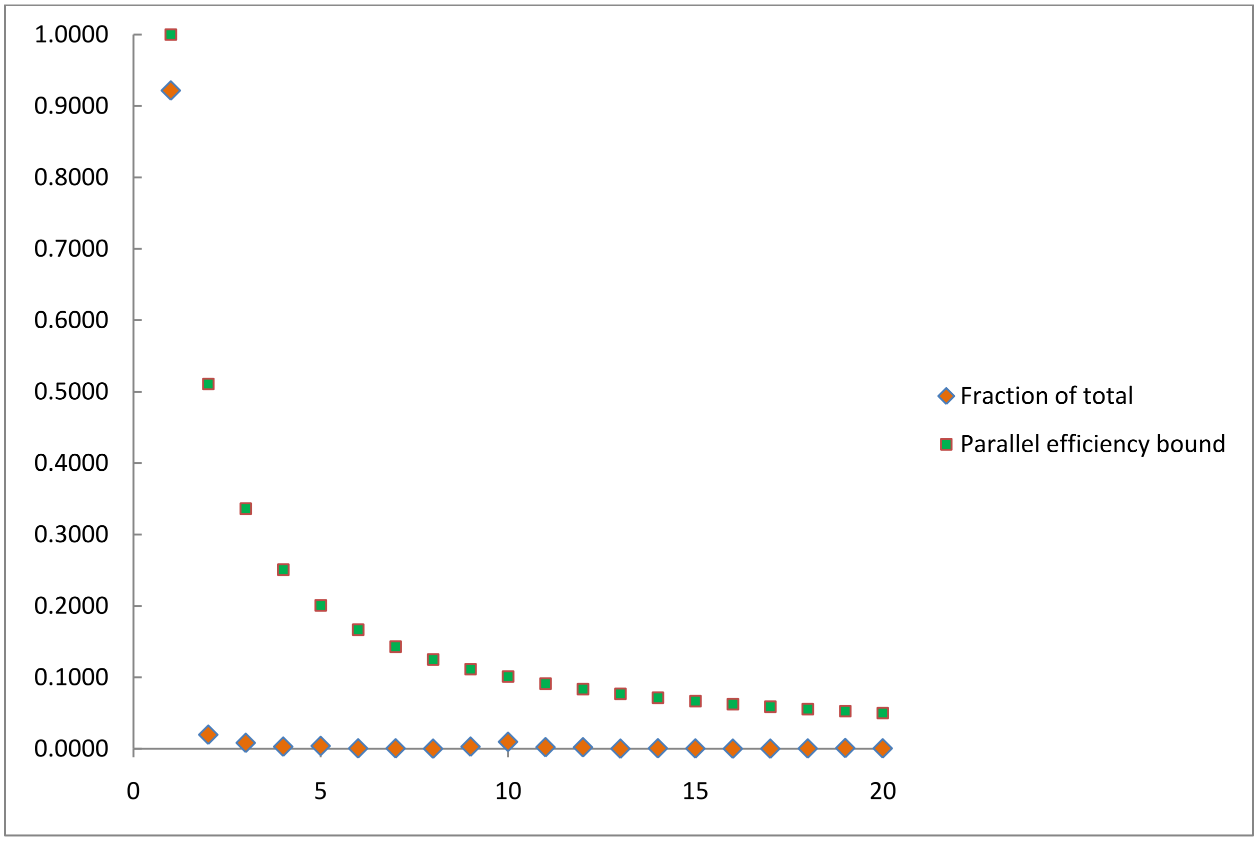


Figure 1 The fraction of the total unique warnings as a function of the analysis run number for an arbitrary ordering of the 20 analysis runs.

From this, the parallel efficiency is calculated as a function of cumulative runs up to run N by taking the total number of unique warnings generated by runs [1, N], divided by N times the number of unique warnings produced by analysis run 1. This rapidly approaches a 1/N curve, which indicates that the efficiency is approximately the same as if one analysis run did all the work, and the other runs just sat idle and did nothing.

the results is close to zero. This is almost certainly an over-optimistic assumption, but even so the efficiency is quickly well below any acceptable level. With 20 processors, the efficiency is about 1/20, which is the same as one processor doing all the work and the other 19 sitting idle.

So, the implication of these tests is that an independent agents design making use of non-determinism in the analyses is not feasible for even small-scale parallelism, and does not improve as the scale increases.

[4] Parallel efficiency is a measure of how much of the available processing power in the parallel array is being used in the computation. An efficiency of 1 means all of the available processing time, while less than 1 means that some available processing power is performing no useful work. Generally for small scale parallelism efficiencies close to 1 are considered good, while for large scale parallelism values above 0.75 are considered good.

6.6 Using multiple processors to increase available process bounds

The Smashproof analysis engine has a large number of adjustable parameters that control the details of the exploration of the intermediate representation. Changing the values of these parameters can have a large effect on the run time performance of the analysis and may also affect the quality of the results. Two studies were performed to isolate any such effects produced by parameters that are amenable to parallel treatment.

The parameters adjusted by the first study are a combination of time bounds and search behavior bounds. The time bounds set limits on how long the analysis can take to complete different calculations. If a time limit is exceeded, the calculation is marked as not useful and any information that might have been generated by it is not available for later use in classifying other program structures. The search behavior bounds include how broad the search will become before truncating further possibilities, the number of procedure inputs that are checked while analyzing a given procedure, and the maximum complexity of allowed logical statements. Checks were run at the baseline level, with all the study parameters doubled, quadrupled, and multiplied by 8. Run times for these analyses were allowed to extend as needed. The same five programs were used for this study as for the non-determinism study above.

A data analysis method similar to the one used above was employed to encapsulate the data. That is, we looked at fractional improvement in the number of unique warnings. However, in this case we were not looking for only the unique warnings in later runs, since a parallel model based on this reasoning would not have multiple agents performing the same analysis. We used the baseline level for the parameters as a standard and looked at fractional improvement compared to that standard. The results showed a best single trial fractional improvement of just 21% compared to the baseline when the parameters were moved to 8 times their base value. Worse, the improvement curve quickly bends to a diminishing returns prediction.

In the chart below, we again plot the fractional improvement in the warning totals of the 5 different program analyses. From this study, there is no reason to think that using large-scale parallelism to support increases in the parameters like the ones explored will be useful.

It may be that we chose the wrong set of parameters to explore, or that a deeper exploration is needed to see the benefits. So, a second set of tests with different parameter settings was attempted.

In the second study, timeouts were extended to infinity, and the bound on search breadth was varied. A different suite of test programs was used for this study[5], as well as using the application binaries instead of the source code.[6]

For this work, there was a question of what to measure. Hand checking of binary disassemblies for errors is not feasible for most real programs, so the study needed a baseline to compare against. We chose the CodeSonar-C analysis of the source files as the baseline. It is not expected

[5] The executables used were gnuchess, irssi, naim, and wu-ftpd

[6] Remember that source code and binary analysis use the same underlying analysis engine. The difference between them is the production of the intermediate representation used in the analysis.

that the analysis results for the source code and the binary will match exactly, but there should be substantial overlap.[7] Measuring that overlap coherently is one data analysis challenge.

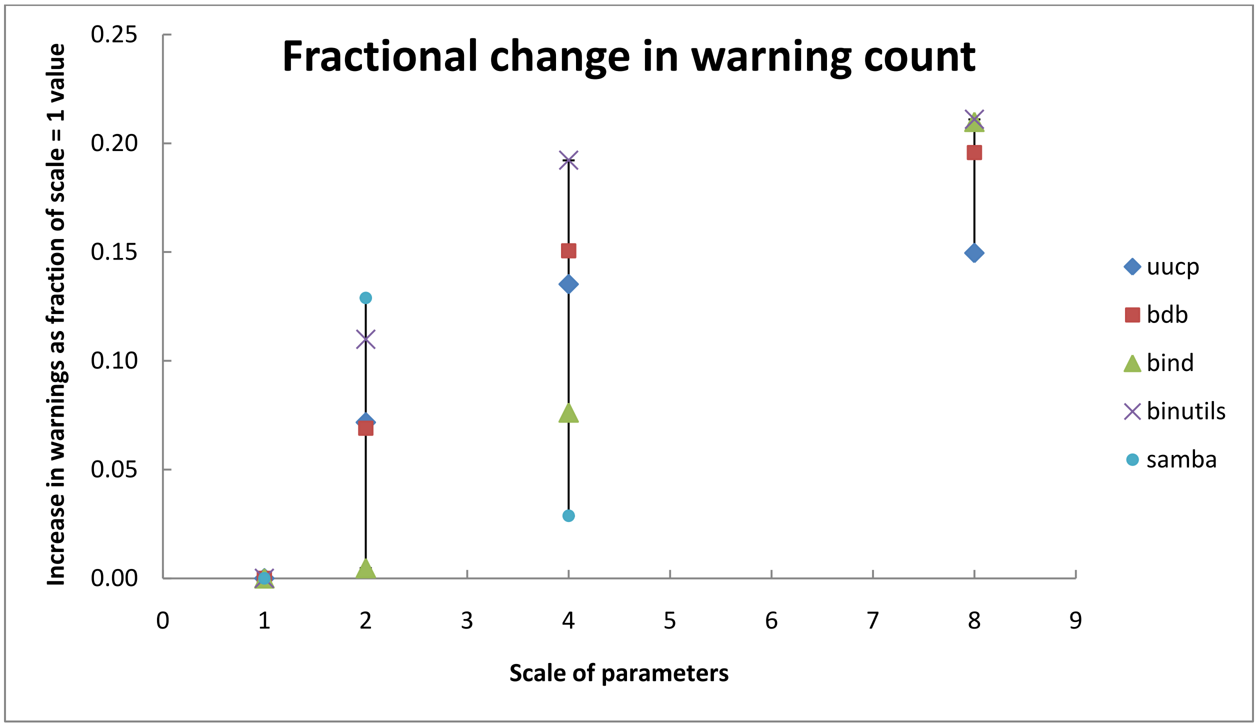


Figure 2 Fractional improvement in the number of unique warnings generated by increasing the bounds on a collection of analysis parameter in CodeSonar. (See the text for the applicable parameters) The bounds were increased to twice, four times, and eight times the current default values and the same code base was re-evaluated. The fractional increase in the number of unique warnings generated is not large, and if the trend established in this study continues for higher values, then it has already reached the point of diminishing returns

Think of the warnings generated as a classical Venn diagram (Figure 3). There are the sets of the warnings generated by CodeSonar-C and those generated by CodeSonar-SWYX. The two sets have some non-trivial overlap.[8] We will call the overlapping section the matching warnings, and measure what fraction that overlap is of the total warnings produced by CS-C. The CS-SWYX warnings that are not part of the matching warnings will be called non-matching warnings, and we will measure what fraction the non-matching warnings are of all of the CS-SWYX warnings.

[7] The IR for the source code is in some ways richer than that for the binary, since it includes information about types and code structure that isn't present in the binary. However, the binary is also superior in some ways, since it is what is actually executed after all of the transformations performed in the compiler and elsewhere. This is an example of the What You See Is Not What You eXecute (WYSINWYX) syndrome. Thus, the name of the GrammaTech tool for analyzing the binary is See What You eXecute (SWYX).

[8] If CS-SWYX has access to the source code, it can find warnings by analyzing the binary and then report the lines in the source code that produce the flaws in the binary. These can be compared for both location and warning class to warnings produced by CS-C to find warnings produced by both.

For each executable, tests were performed at different settings for the search breadth bound. The values used were the baseline value and 2, 4, 8, and 16 times the baseline. Included in this study was a repetition of the measurements at the baseline and 8 times the baseline to estimate how much of the variation in results was due to non-determinism instead of changes in parameters. As was done above, the results are presented for each of the different test programs for each case. The match and non-match changes are measured in percent of total. Additionally, the run times for the trials in the suite were measured, and again the median change in run time is reported – measured in minutes (cf. Figure 4).

These results show that although there may be a substantial run time cost for the adjustment of the parameters, there is no consistent improvement in the quality of the results.

Taken together, the two studies of this section indicate that the combinations of parameters explored do not provide a basis for a viable large-scale parallel architecture.

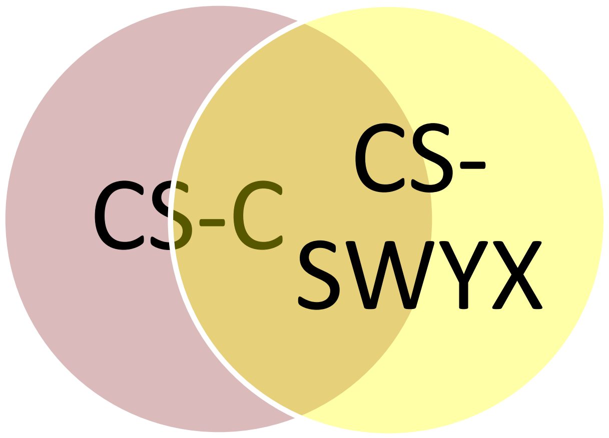


Figure 3 A schematic representation of the overlap between warnings produced by source code analysis, and warnings produced by binary executable analysis. Note that the source code analysis has access to information about variable type, code structure and other things that are not available in the binary executable (because the compiler does not place that information in the binary). Similarly, the compile performs many transformations that are not available in an analysis of the source code.

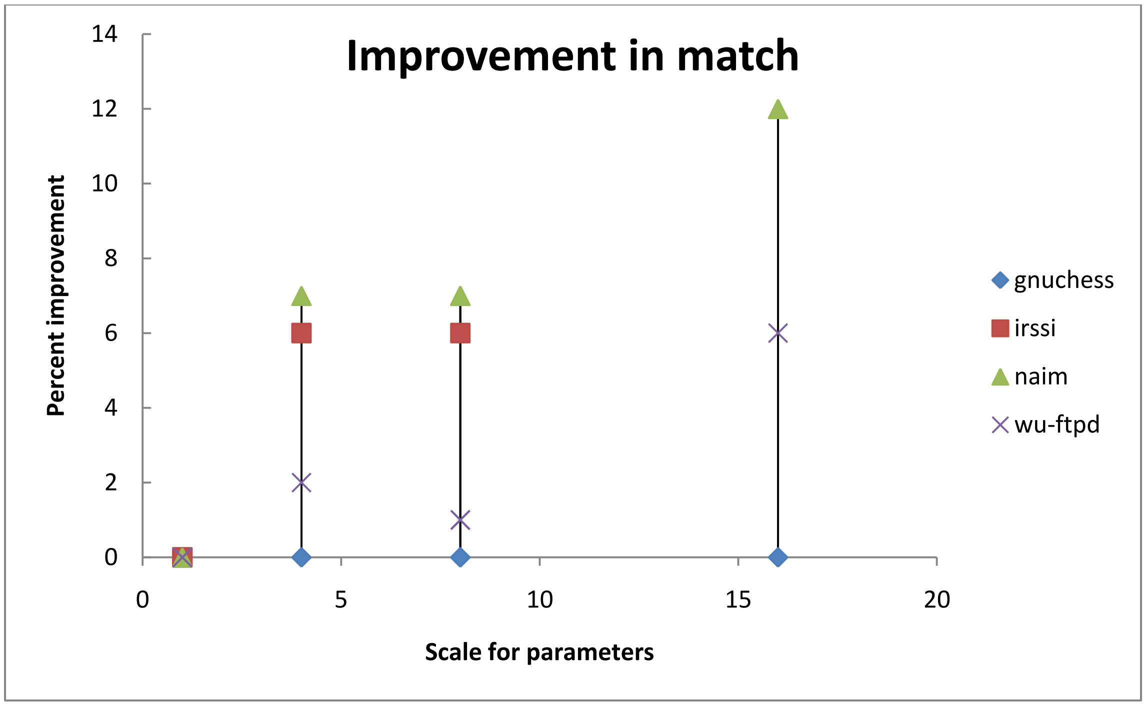


Figure 4 Improvement in the fraction of the generated warnings from binary analysis that match those generated by source code analysis as a function of the increase in the affected parameters. The dependent axis shows the fraction as a percent of the initial value. Multiple values are reported for an increase of eight times because two different runs were performed at that value to estimate the effect of non-determinism on these results. Note that the non-determinism effects are of the same scale as the increased parameter effects.

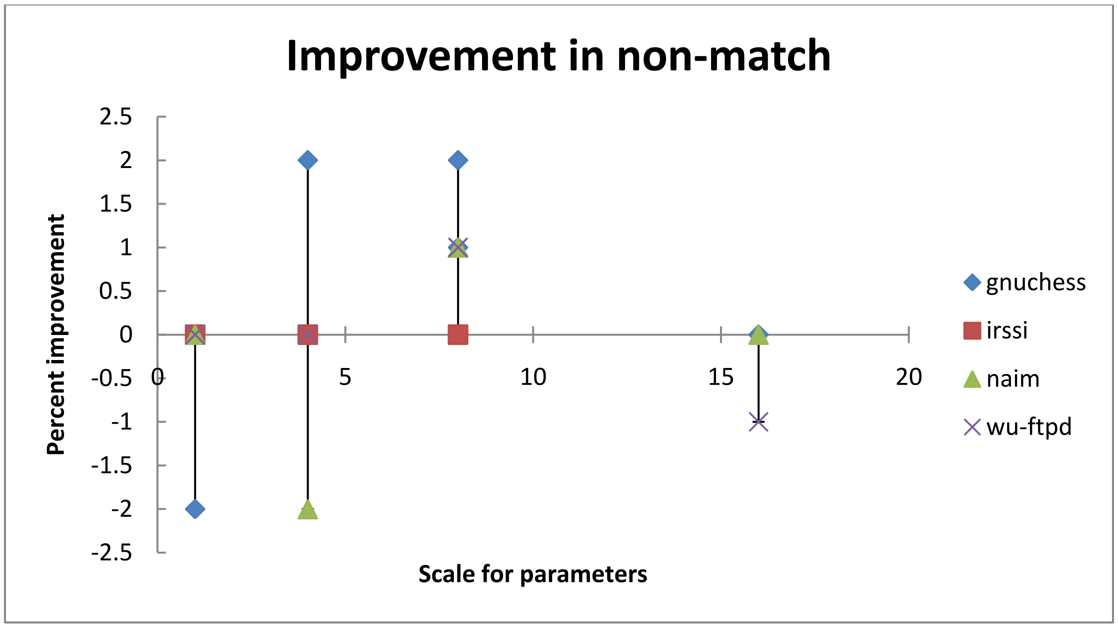


Figure 5 Improvement in the fraction of warnings generated only by the binary analysis as a function of the increase in the parameters describing the algorithm. As before, the dependent axis shows the improvement as a percent of the original value.

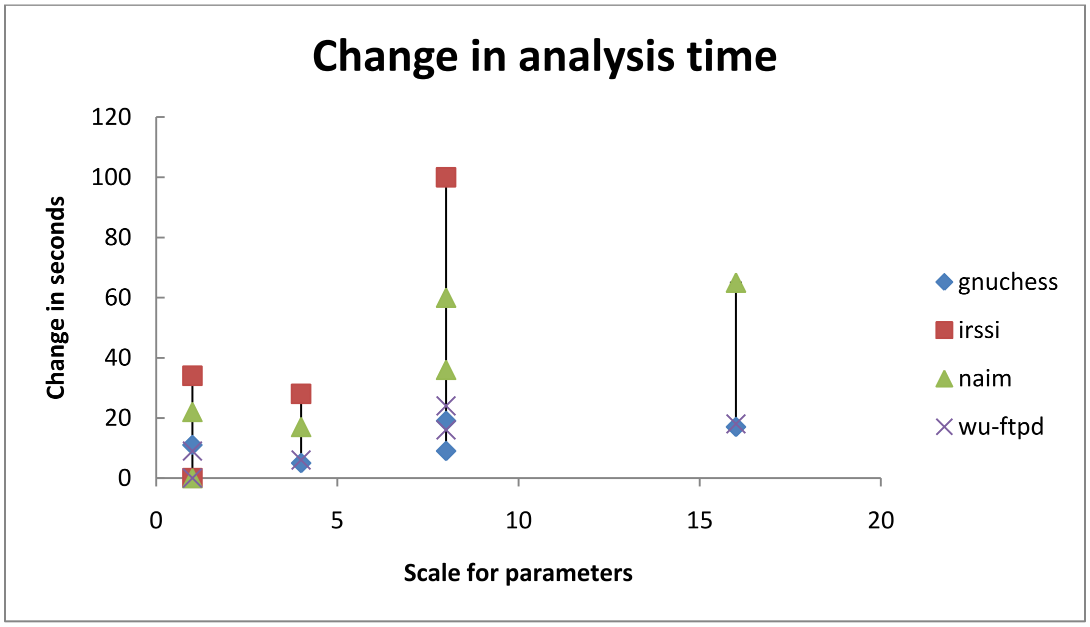


Figure 6 Increase in the execution time as a function of parameter values. This is not a conclusive set of data, since the variation based on non-determinism is similar in scale to the increase as the parameters increase. However, a trend toward increasing time is consistent with this graph.

6.7 Exploring parameters for possible combinations that strongly affect result quality

One further study was performed on the parameters that describe the details of the exploration performed in the analysis. The Smashproof engine uses dozens of such parameters internally, so the curse of dimensionality implies that there is no practical way to explore the whole space in detail. Instead combinations were selected based on expert understanding of the parameters' actions in the analysis. This is an imperfect selection method, since in a complex analysis it is usually true that intuitive perceptions of relative influence of different factors in the analysis are unreliable. However, aside from a very extensive and expensive exploration, this approach provides some useful information about a very hard problem.

Understanding the meanings of all the parameters selected would require a detailed exposition of the analysis algorithms used inside the Smashproof engine. However, the lesson learned from this study is simple enough.

The parameters that had the most effect on result quality were all related to how information about the structure of the program on the large scale was shared throughout the analysis. So, the analysis of function A improved when it learned more about what was going on in function B that was physically separate in the code but interacted with A in some slices through the code.

In terms of parallel and distributed programming, this means that the results improve when the communication of partial results between different processes increases. However, increasing communications between processes in a parallel or distributed computation is very expensive, and so cannot be done on large scales.

6.8 Conclusions from this study

The result of this study is a fairly strong conclusion that the CodeSonar static analysis engine as currently constructed is not amenable to large scale parallelism, and in fact, that the current parameter settings are reasonably well tuned for the current design. Incremental performance gains may be possible with a finer grained approach to parallelism applied to a multiprocessor or multicore single box architecture, but qualitative improvements in capabilities based on grid or volunteer computing designs are not feasible. Any such approach, if possible at all, would require a broad rearchitecting of the engine and could not be performed inside the scope of this project.

This does not imply that improvements in CodeSonar are not possible, merely that brute force (more processing power) extensions to the current design provide little gain. Instead, improvement efforts should address the basic logical representation of the program and the way in which conclusions about warnings are drawn. No experiments that changed the logical encoding were performed here. Those representations and reasoning methods currently include soundness and completeness approximations that enable good performance in reasonable time. Future improvement efforts will focus on this area, rather than parameter tuning of the existing design.

7 Distributed concolic execution

The second technology that we chose to assess in this project is concolic execution. GrammaTech has an existing concolic engine (called GraCE) that we used for this purpose. An introduction to concolic execution of subject programs is provided in section 7.1.

Our assessment indicated that concolic execution is very amenable to parallel execution. Consequently we chose that technology for in-depth analysis. We investigated the following questions:

- What parallel designs might be used for GraCE? Which are the preferred ones? From this study we chose three to implement.

For those three designs we measured the parallel speed-up as a function of various design parameters:

- How much additional work is performed (and how much time is wasted by communication needs)?
- How much useful (non-redundant) work is a performed when multiple processors are used without coordination?
- What is the communication overhead and how might it scale to larger systems?
- Is there an improvement in the quality of program coverage generated by concolic testing when multiple, parallel processing engines are used?

As part of answering these questions we also needed to improve GraCE's robustness (section 9), implemented a validation mechanism for its underlying semantics (section 10), and implemented a test and performance measurement infrastructure (section 8).

7.1 Description of concolic execution

This subsection provides background information. It describes the technique of concolic execution and its relationship to other static and dynamic analysis techniques.

7.1.1 Distributed Dynamic and Hybrid Analyses

One of the more effective techniques for discovering software vulnerabilities is based on a dynamic technique called *fuzzing*. Traditional fuzzing works by feeding a program random inputs and observing what breaks [66]. Newer incarnations are more sophisticated. Random inputs may be constrained to ensure that they are at least well formed [66]. Non-obvious or non-traditional inputs may be fuzzed, e.g., by modeling periodic disk failures [4].

The success of fuzzing in finding vulnerabilities suggests that it could also be used for finding malicious backdoors or time bombs. By leveraging the spare cycles of a large community, fuzzing could become much more effective. However, it is also easy to construct a malicious payload that has an exponentially small chance of being discovered by purely random fuzzing.

Recently, new techniques were introduced that improve upon fuzzing by adjusting the input based on a combination of static analysis and observed behavior. Because these techniques guide fuzzing based on analysis of the code, they are harder to evade and therefore better for eliciting potentially malicious behavior. However, they also can benefit from increased computational resources. In Section 7.1.2 we describe *concolic execution*, also known as “smart fuzzing.” In Section 7.1.3 we describe the DASH algorithm, an approach to “verification by testing,” that is loosely related to concolic execution and fuzzing, but more goal-oriented.

7.1.2 Test-Case Generation by Concolic Execution

Concolic execution combines *conc*rete execution (used in dynamic analysis) with symb*olic* execution (used in static analysis) [72]. It has also been described as *directed automated random testing* [47] and *whitebox fuzzing* [48]. The technique starts by generating a random input for the program (or module) under test. The concrete (actual) execution of the program on the random input is observed. As the concrete execution proceeds, a set of symbolic constraints is recorded that summarizes what must be true for the execution to proceed along the observed path.

At the end of the execution, any solution to the constraints should provide a (usually different) input that would cause the program to follow the same path, execute the same sequence of instructions, and decide all branches in the same way. By modifying the constraints before they are solved, it is possible to get an input that follows a similar, but slightly different path. Specifically, it is possible to generate a (test input leading to a) path that deviates from the original observed path as follows: truncate all of the constraints for points following a branch point, and negate the constraint associated with the branch point. This approach is used to “expand” the initial random input that covers a single execution path into a test suite with extensive path coverage. (The fact that concolic execution achieves high *path* coverage is important, as shown in Figure 7.)

```
int foo(bool b, bool c) {
  int x = 5;    // A
  if( b )       // B
    x = x - 5;  // C
  if( c )       // D
    x = x + 10; // E
  return 10/x;  // F
}
```

Figure 7 Only one input (foo(true,false)) elicits the division by zero at F. Complete code coverage (i.e., A, B, C, D, E, and F are all tested) and complete branch coverage (i.e., B→C, B→D, D→E, and D→F are all tested) can be obtained (*e.g.*, with foo(false, false) and foo(true, true)) without eliciting the error. Concolic execution achieves complete path coverage — all paths through the code are covered — by generating all four possible inputs.

We demonstrate concolic execution on a trivial C++ function:

```
bool bounds_check(unsigned int x) {
    x = x + 100;     // (1)
    if( x < 1000 )   // (2)
        return true; // (3)
    return false;    // (4)
}
```

Let execution begin on a random input x=201,056. The execution begins at statement (1) in the concrete (actual) state x=201,056, and the symbolic state is true (*i.e.*, unconstrained). After statement (1), the concrete state is x=201,156, and the symbolic state is described by a single constraint $x_1 = x_{in}+100$. Concrete execution proceeds to (2), which evaluates to false, causing execution to proceed to (4). The concrete state remains unchanged during these transitions, but the symbolic state is updated by conjoining another constraint indicating the decision of the branch at (2). This yields the following constraints for describing the symbolic state at (4):

$$x_1 = x_{in}+100 \wedge x_1 \geq 1000$$

At this point, the execution ends. Note that any solution to the symbolic constraints that describe the symbolic state, also known as *path constraints*, would cause execution to follow the same path, and that the initial random input is one solution to the path constraints. Concolic execution proceeds to search for an input that will cause a different choice at the last (and in this case only) branch. The constraint from this branch is negated, giving the constraints:

$$x_1 = x_{in}+100 \wedge x_1 < 1000$$

A random solution (say x_{in}=750) is chosen, and a new round of concolic execution is performed. For this example, two rounds are sufficient to achieve complete path coverage, and the process terminates. Note that fuzzing has a very low probability ($1000/2^{32} < 3 \times 10^{-7}$) of generating an input that exercises line (3).

One technical detail for performing concolic execution is how to represent and solve the path constraints. On one hand, the constraints must be easy to solve; on the other, they must be expressive enough to capture the program's behavior. In many cases, the path constraints are based on linear (or affine) relations, such as:

$$v_1 \bullet \frac{k_1 v_2 + k_2}{k_3}$$

where k_i are constants, v_i are variables, and ● represents one of the relational operators =, ≠, ≤, and ≥. [47] and [72] both present algorithms for representing and solving path constraints based (in part) on linear relations.

In the event that the constraint language (e.g., affine relations) is *not* expressive enough to represent the program semantics (e.g., due to a non-affine computation), concolic execution "borrows" from the concrete state in order to simplify the constraints [47]. This may cause the concolic engine to miss some execution paths that are possible (because it over-tightens the symbolic constraints to the specific concrete execution it has witnessed). However, it will always be able to make forward progress and does not get stuck with expensive (or even undecidable) analysis problems. In effect, concolic execution uses the different strengths of dynamic and static analysis to offset the weaknesses of the other.

Concolic execution was the primary focus of this research effort. One approach to identifying malicious behavior is to try to elicit the behavior in a sandboxed environment where it can be observed. Concolic execution is a superb tool for this purpose because it provides such high coverage of the code. (It also provides some control over how "coverage" is defined; it can be as simple as instruction coverage, or as complex as can be specified in the constraint language of the solver.) However, the execution space of real-world programs is often extremely large — too large to ever be completely covered (for many useful definitions of coverage), even by concolic execution. This means that the effectiveness of this use of concolic execution is in direct proportion to the amount of computational power we allocate to it. We believe that by using computational communities and concolic execution, we will greatly enhance our ability to expose malicious behavior (as well as to identify unintentional program flaws).

7.1.3 The DASH Algorithm

The DASH algorithm could be thought of as goal-directed concolic execution, although the algorithms share only a single step [20]. During each round, the algorithm first looks for a path in the abstract program representation that leads to an error state (or a security violation). It then attempts to grow (in a concolic-like fashion) from one of the concrete executions it has previously observed towards the abstract error path it has found. This has one of two results:

1. the solver produces a new test that has an execution path that is at least one step closer to the abstract error path; or,
2. the constraint solver fails to find a suitable input for the desired path.

In the latter case, DASH refines its abstract program representation based on analysis of the failure in the hope of eliminating infeasible paths from the abstraction.

The entire algorithm iterates in this fashion until one of three results is reached:

1. An input is discovered that triggers one of the errors that DASH was searching for.
2. Abstraction refinement results in an abstract program with no error paths. In this case, the abstract program serves as proof that the program is error-free.
3. The algorithm times out. In this case, the algorithm will still produce a test suite with good coverage of the subject program.

As with concolic execution, we are interested in using DASH to flush out malicious behavior. Compared to concolic execution, DASH may be able to employ more sophisticated coverage metrics (for flushing out backdoors). It might also be used to look for specific malicious behavior (e.g., opening unauthorized ports).

The DASH algorithm is easily adopted for distributed computation, in several ways. One approach is to assign each community member a different goal (e.g., coverage criterion or malicious behavior) and ask them to run the DASH algorithm (unmodified) for that goal. Another approach is to assign community members different strategies for generating the (partly random) test inputs that are used to refine the abstract model. One input-generation strategy may cause DASH to get stuck in an endless refinement loop while another strategy may lead to a different refinement-loop trap, or none. It may be useful for a central controller to coordinate the best results from all of the parallel algorithms. For the Safety in Numbers project we did not have time to implement parallel versions of DASH – we confined our work to GraCE.

7.2 Possible architectures for a parallel GraCE

The basic work-flow of GraCE is simple: given an input, analyze the path taken by an execution using that input, and determine alternate inputs that execute along other program paths. This is conceptually very parallelizable: each input can be analyzed independently, producing new inputs to be subsequently analyzed. The only danger is that separately generated inputs will exercise the same program path and produce other duplicative inputs. That concern is discussed below.

There are several methods to parallelize the existing serial concolic engine; we identified seven large-scale architectures, eight variants on those architectures, and a combinatorial explosion of mixtures of ideas from two or more. In this process, we have identified costs and benefits associated with many of them. However, it is not practical to explore all of them in depth.

Instead, we are identified a smaller subset that appear promising on the surface and developed tests that could be implemented and analyzed with reasonable effort; this activity provided more dependable evidence on which choose the prototypes to create. In each instance, an architectural method was identified as promising and tests were designed to explore the central questions regarding the utility of that method. The tests were developed and performed before deciding to build a prototype; only if the results were favorable did we implement that design.

The architectures and variants we considered are enumerated below.

7.2.1 Central work queue of input tasks

In a typical implementation of a concolic testing engine a set of program inputs is used to drive a concrete execution. The trace from that execution is analyzed, and logical statements are generated that describe the conditions that lead to branches that have not yet been executed. These logical statements are solved to produce one or more sets of program inputs that will explore these other execution paths. The input sets are placed in a (possibly prioritized) work queue and retrieved to drive later concrete executions.

One obvious parallel architecture is to build a master process that holds a common queue of inputs that is used by many different analysis engines. When an engine needs an input instance, it draws from the top of the queue. When the engine generates new input sets, they are inserted into the queue.

This is a fairly coarse-grain parallel architecture and is only useful if the queue population can support the supply of worker engines throughout much of the analysis. A useful test is to log the size of the queue throughout the analysis as various programs are executed.

The effectiveness of this technique will also depend on how compact the input sets are. If the input sets are too large, then communication overhead may become prohibitive. This has the potential to lead to contention between engines when communicating with the centralized work queue. Careful design of the communication protocols and use of distributed queues would ameliorate these problems.

7.2.2 Central queues of tracing, resolution, and input tasks

A finer-grained parallelization leverages the fact that the workloads can be refined into steps to generate traces, to analyze traces, and to resolve the logical statements to generate new sets of inputs for later concrete executions. In this design, a master process holds queues for tasks of all three kinds, and worker counts of the three sorts are balanced such that the number of tasks in each of the queues is approximately constant during the analysis.

This requires substantially more redesign and code change than the previous model, but offers potential load-balancing benefits. It also has more communication overhead than the previous model; there is the potential problem that this communication will include very large packages of data to be sent from one worker process to another because the package of trace and logical statement data from an execution is sometimes large.

Measurements of interest include the distribution of communication package size and the relative imbalance of process time spent in the different phases of the analysis. The code was not instrumented to measure these.

7.2.3 Randomized independent agents

It is possible to perform the analysis with no runtime communication at all. Instead, a collection of concolic engines perform independent analyses; after they have reached some boundary the analyses stop and the results from the separate runs are composed together into a single unified result set. This is the simplest to program, since almost no changes are needed to the existing engine at all. There is a need for a script that starts the processes, and for a random seed argument that is used to start each engine in a different state. At the end, a procedure is needed to combine the results from all engines into a single set of results without duplicates.

We need to determine whether such a method is acceptably efficient in exploring the space of possibilities and how much duplication of work happens because there is no coordination among engines. Tests were designed to estimate this effect and the results are presented in section 7.5.

7.2.3.1 Variant 1

In this design variant, the concolic engine is modified so the primary exploration uses a random choice of inputs ("fuzzing"), but the fuzzing is monitored to determine when it is mostly exploring already-explored program paths. In such cases, concolic execution is applied to a fuzzed set of inputs to break out of the over-explored set.

This is a variation on typical fuzz testing and concolic execution suggested by Majumdar and Sen as "Hybrid Concolic Testing" [92]. It would require some modifications and the addition of the monitoring process to determine when concolic evaluation is needed. However, it is likely to avoid efficiency issues that are a concern otherwise.

7.2.3.2 Variant 2

It is not required that the distribution of random input be uniform. Another possibility is to use biased distributions for the different engines, where each engine is biased to prefer some subset of the possible inputs. A good choice of biasing makes it less likely that different engines will reproduce the same testing work. However, making such a choice implies substantial knowledge of the application being tested, and poor choices can lead to reduced coverage of the application possibilities.

7.2.3.3 Variant 3

The current design of the engine starts from a trace of a concrete execution, generates conditions that flip all the branch choices taken during the execution; it then tries to solve those conditions to find inputs that explore different branches. Instead of working with all of the branches, it could use some random subset of the branches in the first concrete execution.

This is a relatively simple modification of the code and is easy to explore once a prototype using randomized independent agents exists.

7.2.4 Divided input space

If there exists some set of different input conditions that is read by almost all runs of the program, and that input can be identified, then the set of all possibilities for that input can be divided among the different processors: each processor would act on a subset of the possible values for that input and would ignore results that would require using other parts of the set.

This method implies some understanding of the possible inputs to the program being tested, and may not be practical for that reason. It also runs a risk that although an input is required for the program, that value of that input could have little or no effect on the execution path. For example, a program that processes communications might require a port number that it uses to make a connection with a remote server. Every 16-bit integer value might be valid as a port number, but changing which value is used for the port does little or nothing to change the behavior of most of the program. An additional issue is that since only some parts of the range might be valid input, any processing nodes that have only invalid possibilities in their range would operate inefficiently. Attempts to mitigate this problem with load balancing and work stealing algorithms greatly increase complexity.

It is difficult to create a useful test protocol for this method. It depends very strongly on the properties of the program under analysis, so any testing structure tells you more about the choices made to develop the tests than about the method. The lack of such a testing method makes this design less appealing than many of the others in this list.

7.2.4.1 Variant 1

With a deeper understanding of the program under test we could make sure that any required input was actually important. However, this again increases the human intervention needed in the testing process to specify what inputs and ranges are the important ones.

7.2.5 Divided execution path space

As seen above, there are several potential problems with dividing the space of inputs among different concolic engines. So, instead of worrying about which inputs are needed, we could divide the space of execution paths.

This technique assigns a number to each processor and uses that number to choose the branch taken at the first few branch points encountered in the code. This is the starting place for the different explorations. For example, if the number of processors is 2^N, the N bits of the binary representation for each processor's id can indicate the true/false choice to make at each of the first N branch points. In this way the processes can be tied to the input needed to determine those branches and only explore variations that affect later branches.

The main issue for this design is that there is no reason to think the first few branches in a program encapsulate important differences in the execution. For example, if the program takes command line input and validates that input, the first few branch points may all lie in the argument validation, so many of the choices lead to validation failure and no further exploration, while only a small fraction explore the majority of the program.

With non-trivial but still small programs it is possible to estimate the effects of the first N branches. This would include a consideration of the likelihood that some of these paths are already consigned to dead ends given the first N choices.

7.2.5.1 Variant 1

Instead of pinning the decisions at the first few branches, fix them at some set of branches randomly chosen from the list of possibilities. This avoids the issue with validation or initialization code.

7.2.5.2 Variant 2

Instead of numbering branches by the order they are encountered in a trace, generate a numbering assignment for all of the branches in the code. Traces that do not hit branches for which they have assigned decisions just ignore that decision value.

Producing such a numbering may itself take some meaningful analysis, however.

7.2.5.3 Variant 3

Instead of using a binary representation of the number of processors, select some larger number, preferably a power of 2, and at least an order of magnitude larger than the number of processors. Keep these numbers in a central queue such that a concolic engine with no other work to do requests a number from the queue and uses that number to divide the execution path space.

This reduces the impact of a single process receiving a set of branch selections that are a dead end. Such a process finishes its current exploration quickly, and then moves on to a different exploration based on a new number from the queue. Using a power of 2 ensures a simple mapping to branch options; having at least an order of magnitude more items in the queue than processes will usually give a reasonable load balance without the need for complex load balancing code.

7.2.6 Goal driven concolic execution

Current concolic engines are exploration driven. The algorithm is simply trying to find inputs that lead to new execution paths, trace those paths, and process the information in the trace to find more new paths; each path is assessed for errors or presence of malicious intent. Another possibility is to drive the exploration with some set of goals. This could be built on the DASH algorithm [20]. Providing different engines with different goal sets would lead to different explorations of the possibilities. A well-designed set of goals would also lead to a more efficient exploration of potentially problematic code segments.

Currently DASH and a related algorithm from Thakur et al. [93] that applies the same ideas to machine code, called McVETO, are themselves research projects. We estimate that substantial work on the underlying processes would be needed before this technique could be incorporated into a concolic prototype. So, at this time it is not a good choice for a prototype system.

There are no meaningful tests of this process that can be done using small modifications of the current engine, since the algorithm used is substantially different and would require construction of a whole new engine.

7.2.7 Seeded with regression tests

If we assume that regression tests for an application are available and that they are written with the intention of exercising important sections of the code, then another useful method may be to start concolic execution runs from regression test inputs instead of from random inputs.

These starting places probably ensure that different engines start exploring different parts of the code, and that the starting places include important parts of the code. However, it also builds the assumptions of the original developers into the testing process and may be very inefficient at finding bugs that are different from those assumptions.

Moreover, it requires access to regression tests. This is a reasonable assumption for a developer working on the code, but not for third parties who wish to extensively test received code before trusting it. In particular, malware developers have no history of shipping regression tests with their products.

A small modification to the existing engine would allow for initial inputs to the program under analysis to be provided by a user instead of being randomly generated. Combining this modification with a few regression test inputs and comparing coverage and efficiency to tests generated by random inputs would be a reasonable test of the viability of this method.

7.2.7.1 Variant

To combat the problem of tying testing to the developers' assumptions, we could use regression test inputs as a base for randomization. This might include replacement of parts of the input with random data, or random mixtures of several different inputs. Of course, this does nothing to address situations that have no tests.

7.2.8 Fine grained parallelization of the engine

Not all parallel architectures are coarse-grained. It is also possible to parallelize individual loops, functions or algorithms. Such designs are most useful when a small number of well-contained segments of the program are responsible for most of the run time and are amenable to parallelization.

This is the most intrusive design in terms of the modifications needed to the serial code. It is also the most difficult to test and load balance. In distributed or grid code, it is even worse, since such a design requires distributing components of the underlying algorithms over many machines and combining the results of the computations with synchronization before moving on. Thus there is a strong potential for high communication and synchronization overhead.

As such, this architecture was not considered for this project.

7.2.9 Combinations

Finally, combined methods that use parts of the previously defined architectures cooperatively are reasonable in many cases. Such combinations come at a cost of additional complexity, and it can be difficult to develop tests for viability for many of them. Furthermore, part of our goal is to clearly assess the merits of individual designs. Thus we will combine different approaches only when the various architectures clearly complement each other. Thus combination architectures are a lower, or at minimum, later, priority for these explorations.

7.3 Selection of designs to prototype

The extensive taxonomy of possible architectures was too large to investigate in this project; the list had to be trimmed to a set of prototype candidates. The criteria for choosing parallel designs include the promise of exploring interesting design spaces, potential for massive parallel implementation, and relative testability of the implementation for correctness and stability.

Based on those considerations, three designs were selected:

- *random initial inputs to a set of independent concolic engines*. This design allows additional processors to be added easily, but the independently running engines may duplicate a lot of work.
- *crafted inputs to independent engines*. This design also allows processors to be added easily, but by using white-box-chosen inputs, we expect to obtain higher coverage and may avoid some duplicated work.
- *a shared single work queue*. In this design a single boss process distributes work among a number of worker processes, who report results as they are completed. No work is duplicated in this design, but the communication overhead will be higher and will need to be measured to determine scalability.

A more detailed description of each is provided below.

7.3.1 Independent engines with random initial inputs

Traditional fuzz testing uses random inputs to test an application. Our first design parallelizes that approach by using randomized initial inputs for many independent copies of the concolic engine. This is a straightforward implementation, and it keeps the work performed by the different engines independent. There is a small communications burden at the start of the analysis when the randomly generated inputs are assigned to the different engines, and there is a communication at the end when the results of the exploration are collected for final reporting. However, in between, the calculation is fully independent and no communications between processes is required.

This is ideal for massive parallelism, since frequent communications impose a need for synchronization and are error prone even in small scale parallelism. In addition, communications are typically slow compared to computation; so, as the number of processes in the cluster increases, the fraction of the run time used for communications increases. Thus, programs that require frequent communications have an upper bound on potential performance gain from adding more processors.

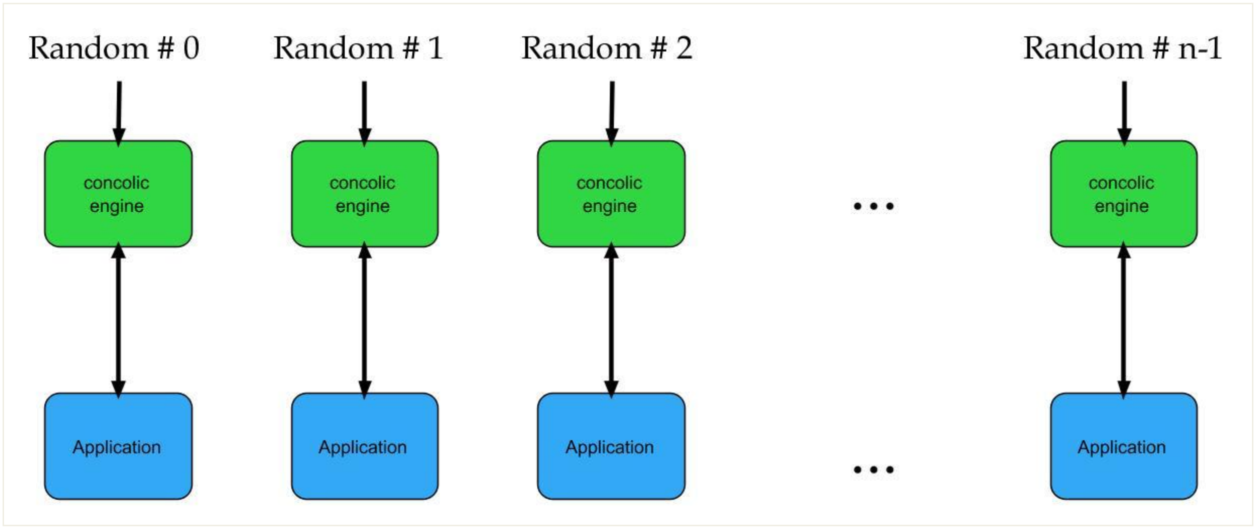


Figure 8 Schematic representation of the architecture of the independent agents with random seeding design

The random independent engines design avoids both of those pitfalls, and makes use of an idea that is known to be useful in some contexts. Randomization of inputs has been used in traditional fuzzing analysis with meaningful success in some cases. In this design, we combine fuzzed initial inputs with concolic exploration of possibilities generated by those inputs in what may be a best of both worlds approach.

However, there are possible problems.

In the current implementation, the random input generation assumes nothing about the structure of the inputs that should be given to the subject application. If the subject includes extensive input validation, almost all of the random inputs will fail the validation; they will then begin their concolic exploration from approximately the same trace. In other words, many of the independent engines will explore the same traces and the exploration will be very inefficient. Only the inputs that happen to proceed beyond the validation phase will explore novel execution paths.

7.3.2 Independent engines with crafted initial inputs

One way to solve the problem of collapsing trace exploration is to provide inputs that are known to explore different parts of the trace space. The crafted inputs prototype does just this. Instead of beginning from random inputs, this engine has a database of inputs that are known to exercise different parts of the code. So, engines that start based on these different inputs will be sure to begin exploration from different traces. These different traces produce different concolic explorations of the set of all possible traces, and so the problem of collapsing traces is at least partially avoided.

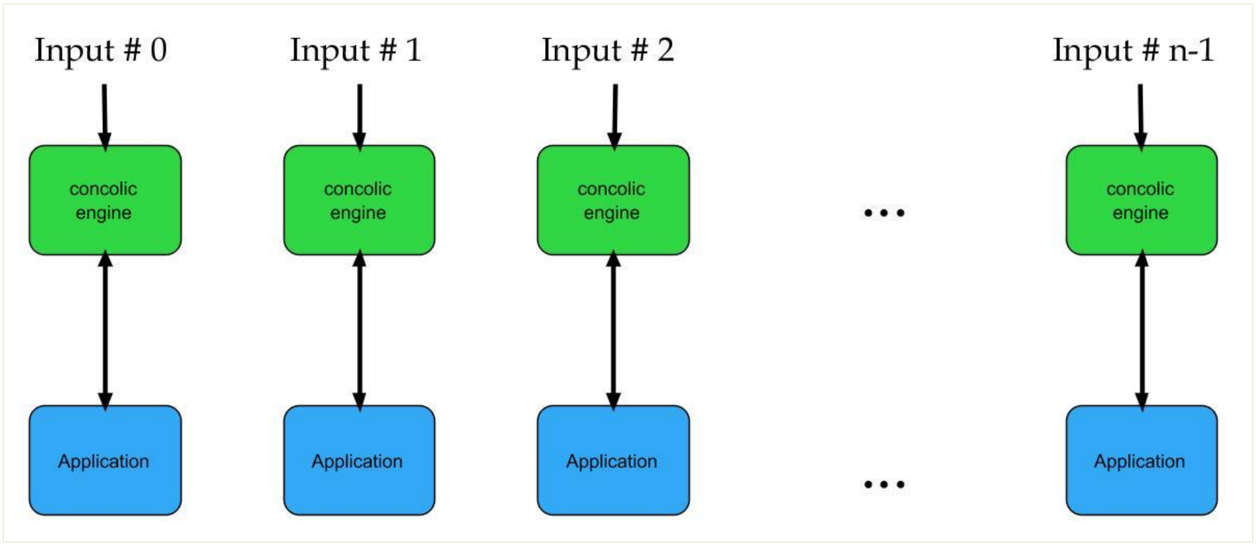


Figure 9 Schematic representation of the architecture of the independent agents with crafted initial inputs design

The initial inputs can be generated directly, but they are easiest to produce when working with a code base that already has a strong testing suite. In such a case, the test suite is built using the developers' understanding of the error prone and important sections in the code. Using those inputs to fill the database of initial inputs then builds the developers' understanding of the code into the testing system. This has both positive and negative implications, however. If the developers understanding of a problematic piece of code is lacking, the test suite probably also does little to explore that code. In fact, since the initially generated traces are the ones from existing tests, the starting places for concolic exploration are likely to be some of the most stable and bug free traces in the code.

This method also is not very useful for third party users of the subject program or for detecting malware injected into otherwise useful code, since it is unlikely there is access to a test suite for either case.

7.3.3 Multiple engines with a shared work queue

A different way to address the collapse of the trace exploration is to assure that multiple engines only redo work if you want them to. This can be accomplished by a central, shared queue. The queue holds the initial inputs needed for continuing exploration of the set of all traces through the subject program. It includes a database so inputs that have already been explored can be suppressed and the exploration can focus on inputs that haven't been tried before.

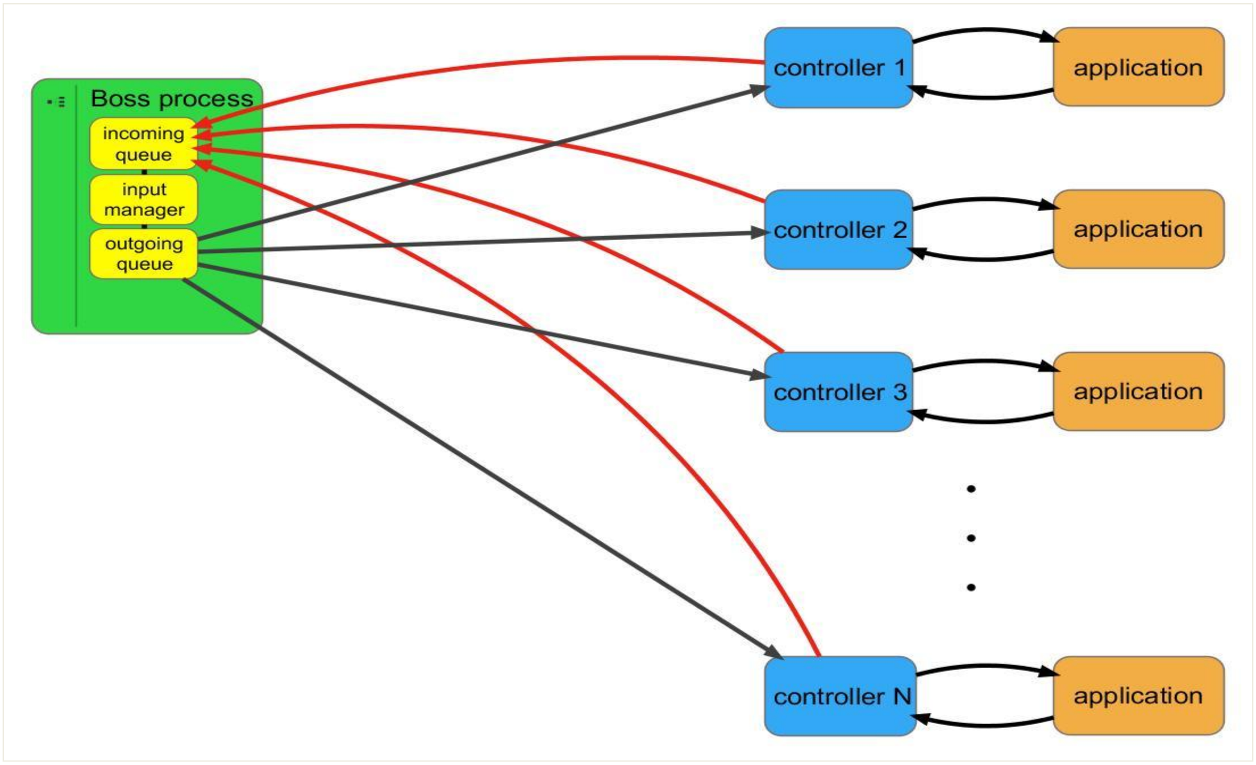


Figure 10 Schematic representation of the architecture of the shared work queue design

This has some side benefits: the database can decide to explore the same inputs a few times, if desired. Re-examining an input protects the integrity of the results in circumstances where not all of the computers in the cluster or grid can be trusted without question. This may be due to hardware issues or to an intentional undermining of the analysis, but several standard ways to guard against it are possible with the centralized queue.

The problem in this design is the need for frequent communications between Boss and Worker. As explained above, this can become a limiting factor in the ability to scale to arbitrarily large clusters. However, hierarchical arrangements of local Bosses and other design changes can mitigate this problem if needed. As is typical, they trade decreased communications overhead for increased computational complexity, but the trade may be justified. Furthermore, for large programs, the work performed will be dominated by running and analyzing the subject program; communications costs will be relatively smaller as the subject program complexity increases.

7.4 Implementation work

We completed the implementation of all three designs within this project and measured the performance differences among the designs. As we applied the tool to larger and larger programs, additional issues with the overall framework became apparent. Some are problems with the GraCE tool itself; these were addressed and are discussed in sections 8-10.

7.5 Assessment of performance results

Extensive collection of data from our parallel prototypes was an important feature of the project. The results reported in this section represent thousands of hours of computer time and are condensed from several gigabytes of collected data. However, it is important to understand that the space explored in parallel performance studies is large and has high dimensionality, so a complete exploration is simply not feasible except for the simplest examples. There are four characteristics that we measured:

- The speed-up in the amount of work performed as a function of the number of processors, the parallel prototype design, and the type of program being analyzed. Note that in our current shared-queue design, one processor out of the N total is devoted to managing the shared queue. Hence we would hope for a speed-up of at most (N-1)/N. However, that master process actually is quite under-utilized when the program inputs are small. For groups of processors of the scale we have measured (up to 20), it would be better in some cases to have that processor perform both queue management and analysis work. One of our goals is to estimate at what scales the queue controller process becomes overworked.
- The increase in the amount of *unique* work performed as a function of the number of processors, the parallel prototype design, and the type of program being analyzed.
- The amount of branch and statement coverage of the subject programs that GraCE achieves. Branch and statement coverage is a classical measure of test case quality. Concolic execution enables exploring a large number of unique execution paths. This in principle creates a higher-quality test suite than just relying on branch and statement coverage. However there may be an unbounded number of paths and a concolic engine will only explore a subset of them, limited by time and memory resources. It is possible that the concolic search engine will focus on a subset of paths that do not, taken together, have good branch and statement coverage. Hence we measure this coverage as a check on the concolic engine's performance.
- The amount of communication overhead that the parallel designs incur. This is an important measure in any attempt to estimate scaling behaviors.

Our conclusions are the following:

- Communication overhead in our examples is generally measurable but quite small and not at all an impediment to performance. Only by explicitly crafting a program could we obtain significant communication overhead. Three characteristics of our program set may overly minimize this problem.

- GraCE is currently not very efficient at analysis. Improving GraCE's performance will raise the proportion of communication in the overall time. However, the native execution time of the subject program will always be a limit. Hence for programs that are large enough to be of realistic interest, we still would not expect large communication overhead.
- We currently only communicate a specific command-line for each new worker process iteration. When file and network inputs are implemented we will also need to communicate the data that constitutes these inputs. This will enlarge the amount of data needing to be communicated.
- The shared-queue controller can always be swamped by a large enough worker set. That size appears to be at least a few thousand workers in the worst case and much larger for typical cases. If that ever becomes a problem a tree structure of controllers can be used.

- Though it varies by program type, the speed-up achieved by using N processors is typically at least 75% of N and approaches N for larger programs for any of the three designs. However, the speed-up is measured more appropriately as the amount of unique work accomplished. For the random-input prototype, this can be quite low, except for programs with a wide fan-out of execution paths and little input validation; the crafted-input design does better, but requires a good understanding of the subject program, which is non-trivial for complex programs. All the work performed by the shared-queue design is unique.
- The principal limitations of these measurements are these two:
    - we have only made measurements up to 20 processors (though for random-input and crafted-input designs, we can accurately simulate any number of processors);
    - the communication overhead will increase as different input mechanisms are implemented in GraCE. For example, when file input is tested, the system must send candidate files (instead of just the command-line) to the worker process.

In doing this study we used a range of programs, as described in the following subsections.

7.5.1 Small, "toy" programs created just for this purpose.

These programs generally have simple code, with just 100-200 lines in size, and compiled sizes of a few KB. The analysis time is dominated (70-90%) by running the subject program, since the path analysis is simple and straightforward. In some cases, the number of paths in the program is small enough that all of the paths can be explored. These programs include the following:

- Cmd_args.005: an "Easter Egg" program that has a sequence of branches that are very unlikely to be chosen at random; the particular Easter Egg output is only triggered by a very specific input. GraCE finds the unusual path without difficulty. There are about 1300 paths in all possible executions and GraCE can explore them all.

- Most of the time is spent communicating between GraCE and the subject program (~80%)
- We measure a ~15 times speed up for 20 processors using shared queue; this gives a parallel efficiency of ~75% for the shared queue prototype
- The program has periods where the shared queue is filling at the start and emptying at the end where most of the processors have no useful work to do (this is the principal effect degrading the speed-up).
- With the random initial inputs prototype, there is no observed speedup for complete exploration of the paths. In fact, even for partial exploration of the paths the speedup is small or non-existent. This is because only rarely does a random initial input explore beyond the first couple of branches in the initial input validation. The explorations for the different nodes quickly collapse down to being multiple copies of the same exploration. The only way to assure full path coverage is to let all the nodes run through to full coverage independently, which means every node is doing all the work of developing full coverage.
- The unique work efficiency for the random initial inputs prototype is then (1/N)*100%, where N is the number of nodes. This is equal to the theoretic worst case.
- For the crafted initial inputs prototype, the initial partial exploration can be very efficient. A well chosen set of initial inputs provides >85% branch and statement coverage in one iteration of each node in the cluster. However, the analysis of the first instantiation of the subject program produces some common results. This reduces the unique work efficiency as the number of instantiations of the subject gets larger. The only way for the prototype to assure 100% path coverage is to let every node run the complete analysis from the different starting places. (This is because the nodes do not communicate with each other, and so cannot compare notes to see if a different node has checked the path produced by a given input. Without this ability, the only available stopping condition that assures full coverage is to allow each node to attain full coverage independently.) This again leads to the theoretic worst case efficiency.
- The branch, statement, and path coverage are all 100% for all the prototypes
- Additional studies were done with the shared queue prototype to produce an estimate of the influence of uncontrolled factors on the speedup measurements. A cluster of 16 machines was run several times to perform the complete analysis, and timings were compared. The results showed a range of parallel speedups from a low of 8.4 to a high of 13.1. Since there is no duplication of work for the shared queue, that is the same as efficiencies from 52.5% to 81.9%; this variation can be ascribed to a collection of factors.
    - The machines used for the performance study are not exclusively dedicated to that use. Other computing jobs take resources and slow down the computation.

        - The machines used are connected via the facility's internal Ethernet network. The entire facility is in the same subnet, so traffic anywhere in the facility will reduce available bandwidth for cluster communications.
    - Such variation would be substantially reduced in a dedicated cluster setting. Also, a cluster designed for HPC would not use Ethernet with a single subnet, but would use a high performance interconnect such as Infiniband with a collection of switches and hubs that minimized the chance of messages from one node colliding with those from another.
- Misc.002: a program that is meant to have very simple computation and so emphasizes the communication overhead. It validates the form of the input, then manipulates the bits slightly.
    - It takes quite a few iterations at the start of the run to build up a decent queue size, so workers spend time idling at the beginning
    - The boss does not block on the queue, so boss time spent on communications is a good measure of how long the communications themselves (without blocking waits) took.
    - The worker processes do block, which explains their much larger fraction of time spent in communication (or waiting for communications).
    - Multiple measurements were made of the speedup for different cluster sizes running the shared queue prototype and performing 1000 iterations of the subject program. The speed up is based on the run time for the shared queue prototype compared to the run time for 1000 iterations of the serial implementation. The speedup measurements are presented in Figure 11.

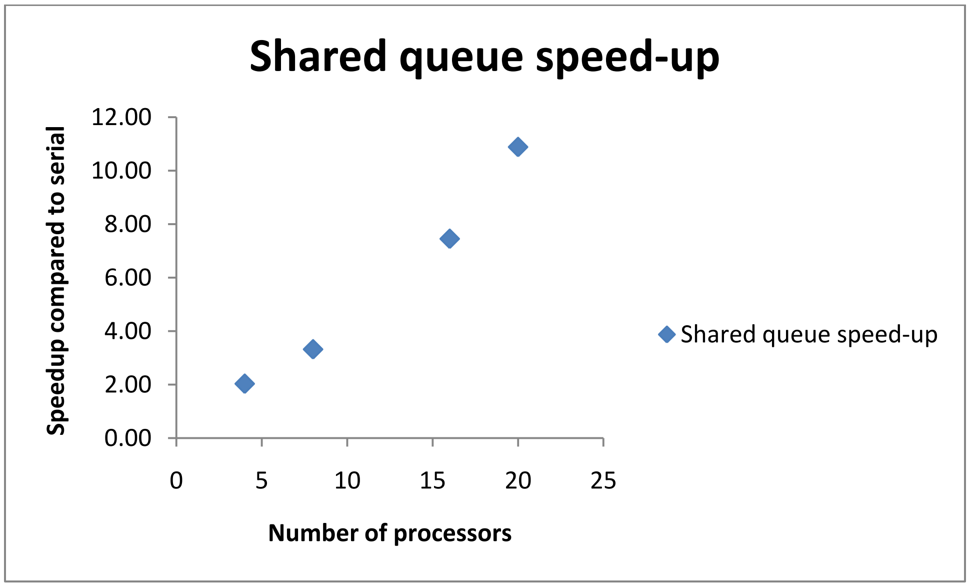


Figure 11: Speedup of misc.002 for the shared queue prototype

From the speedup measurements and a measure of how much work is unique, it is easy to determine the efficiency - that is, how much benefit was derived from each new processor added to the cluster. In Figure 12, we see that the efficiency is approximately 50%. Unfortunately, the effects of uncontrolled performance factors are large compared to any cluster size scaling effects for the cluster sizes investigated. This makes it impossible to extrapolate the effect of cluster size on the efficiency and determine a cluster size where the efficiency is no longer acceptable.

Because of the worker time spent blocked on communications and the very short run and analysis times for the subject, this example does not scale very well with added processors. Efficiencies are under 60% in every case for cluster sizes of 4, 8, 16, and 20. They are fairly consistent for the different sizes, however.

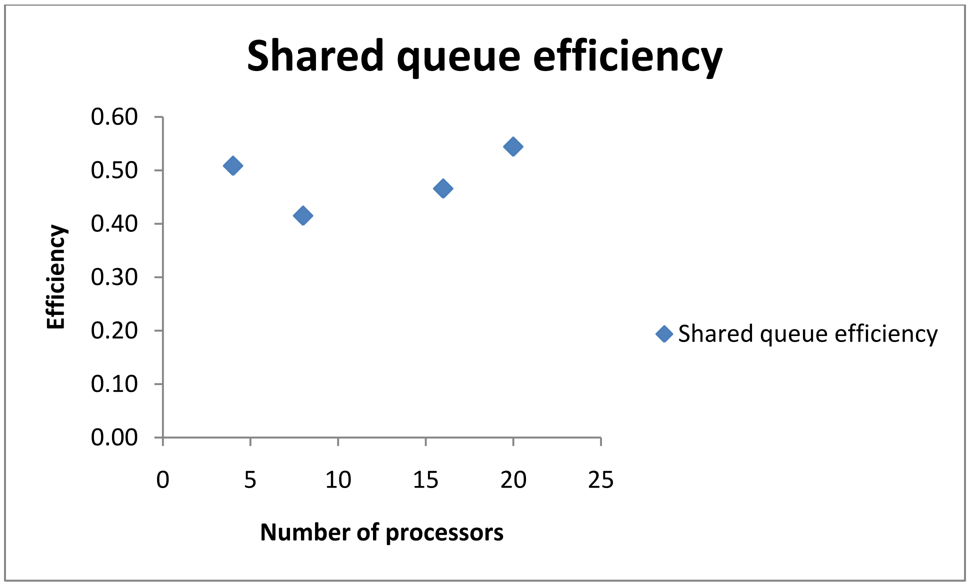


Figure 12: Efficiency of misc.002 for the shared queue prototype

Figure 13 shows the percent of run time as a function of number of processors for this program. These data show that the communications overhead is very small (<0.5%), even in this nearly worst-case program, but the time spent blocking by workers while the queue builds up is a concern. It appears approximately quadratic in the number of processors, so the blocking wait time would start to dominate the run time at around 100 processors (for 6000 iterations); the effect reduces as the iteration count grows. It is harder to estimate the actual communications time from this data. The noise in the measurements (due mostly to not using dedicated machines or network connections) is comparable to the effect being measured. However, if we assume a linear growth[9] in the communications time with the number of processors it would take approximately 300 worker nodes to reach 10% of the run time spent on communications cost.

[9] As we will see below, linear growth is probably not a good assumption.

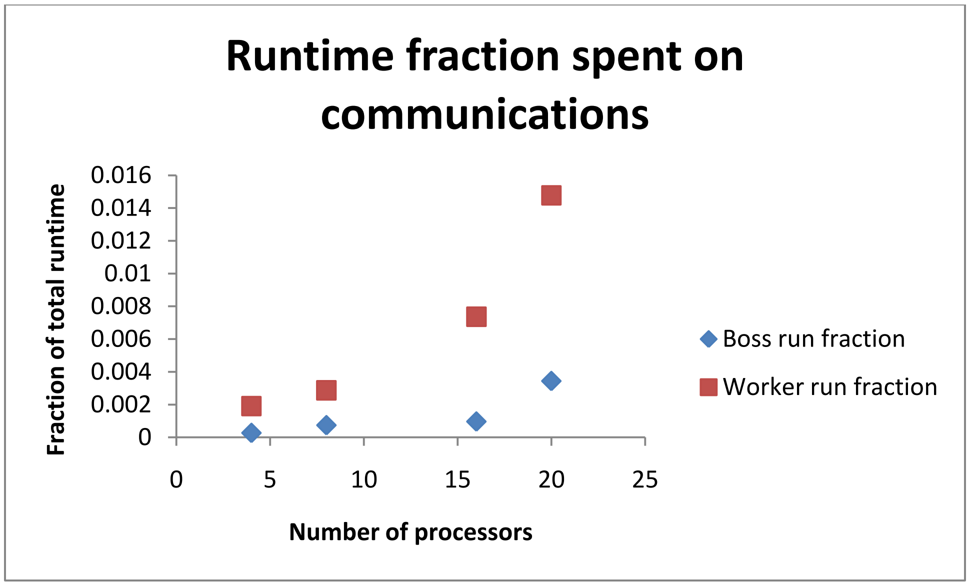


Figure 13 [Misc.002] Fraction of total run time spent on communications by Boss and average worker, as a function of the number of processors in the cluster. The Boss uses non-blocking communications, since there might be other work to do, while the workers use blocking communications because they are requesting their next work item from the Boss and don't have other things to do until receiving that work. As such, the difference between the Boss and Worker timings provides an estimate of the time spent blocked by an average worker.

The primary issues are probably that the communication incurs some indirect costs in extra object constructions during the serialization and deserialization needed to transmit inputs, as well as having one processor in the queue dedicated to coordinating communications while not doing any analysis work. A more complex design could reduce the penalty for the second issue in cases where the communications load is small, but there is little that can be done about the first issue. The effect is reduced for even slightly more complex toy programs, however (cmd_args.005 is close to 80% efficiency, for example).

Running independent agents with random initial seed inputs for each engine produces very poor scaling in terms of unique work done, as shown in Figure 14. With 5 nodes, approximately 40% of the work is unique; however, with 40 nodes, it is about 20%. So, even with small clusters, in this example the efficiency is very poor, and as the cluster grows the efficiency drops rapidly. This is a program that validates input at the start (it checks for the existence of argv[1], and then checks to see if it can be interpreted as an integer.) Poor performance is expected in this case for random seeds.

From the raw numbers graph, another fact becomes obvious. In a test case with a large number of paths, the raw numbers of paths generated and unique paths generated are not the interesting quantities. The interesting information is in the ratio of unique paths over total paths. This fraction tells how efficient the process was at producing new information during the run, and so can be sensibly compared to the efficiencies produced in analyzing the shared queue prototype. This is a much more valuable measure than the total work for another reason. The random initial inputs and crafted initial inputs prototypes both have close to zero communications load during an analysis of a subject. Because of that, the total work done is almost exactly the work done by a serial process times the number of processors. However, the lack of communications means that the different processes in the cluster can't coordinate their efforts to avoid duplicate work. It is this duplicate work that prevents the net efficiency from being ~100%.

Figure 15 presents this efficiency measure for the random initial inputs and crafted initial inputs prototypes.

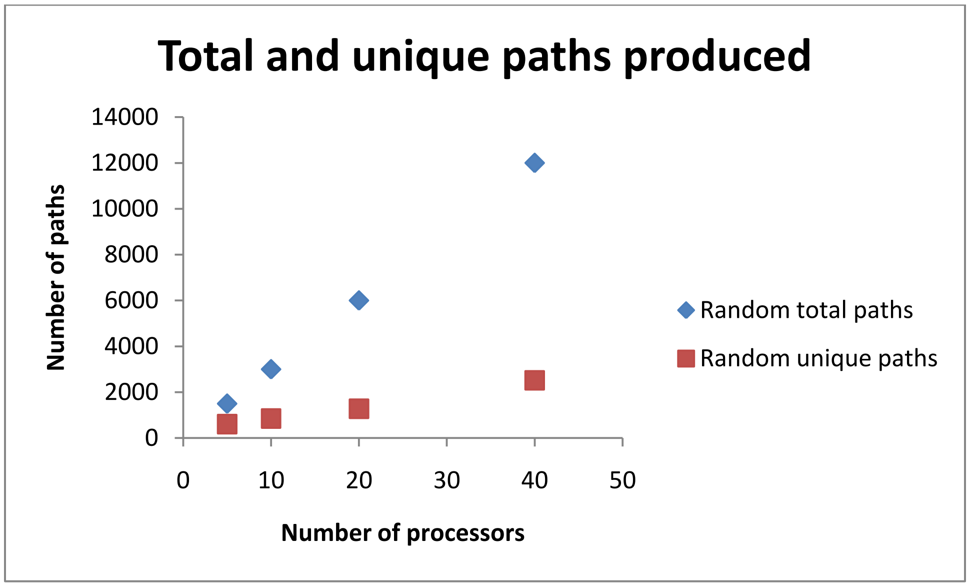


Figure 14 The total number of paths produced by a cluster of independent agents starting from random seeds, and the number of those paths that are unique, both as a function of the size of the cluster.

(The physical cluster in use has 20 nodes, but since the only distinguishing characteristic of different nodes in this architecture is that they start from different random initial inputs, the physical cluster can be run multiple times and the results produced can be merged to show the results from an equivalent larger cluster)

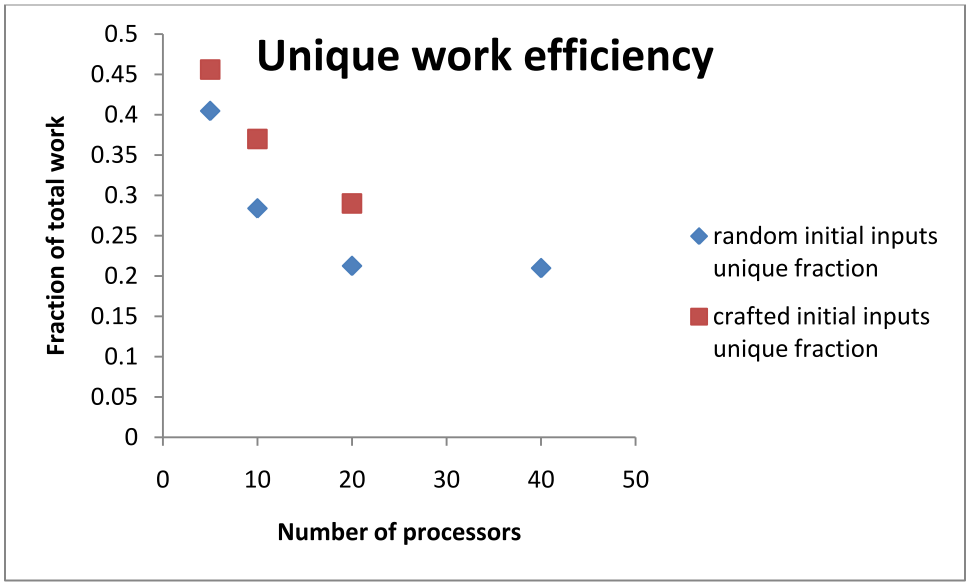


Figure 15: Unique work done by the random initial input and crafted initial input prototypes as a function of the number of processors in the cluster. The measure is just the number of unique execution paths generated divided by the total number of execution paths generated by the measured cluster. Uniqueness of the execution path is determined by whether the sequence of blocks traversed in the trace is unique.

- Misc.003: a program that is designed to have very simple computation and no input validation, and so emphasizes the communication overhead (even more than misc.002).
    - The lack of validation reduces, but does not eliminate the overlap between different instances of the random initial inputs or crafted initial inputs prototypes. Even designing a program with no input validation, each independent processor explores paths with illegal numbers of input arguments, so there is still some overlap in work. The only way to avoid all of such overlap is to design a search heuristic that completely deprioritizes paths that are likely to be common to all independent processes.
    - The random initial inputs and crafted initial inputs produce acceptable unique work efficiencies for small numbers of processors, but not for larger numbers, as discussed below.
    - Extensive studies of the response of the shared queue prototype to additional communications loads were performed with this subject program. They are described in detail below.

The unique work fraction for the random initial input and crafted initial input prototypes is shown in Figure 16. The data displays one of the problems of the crafted initial inputs

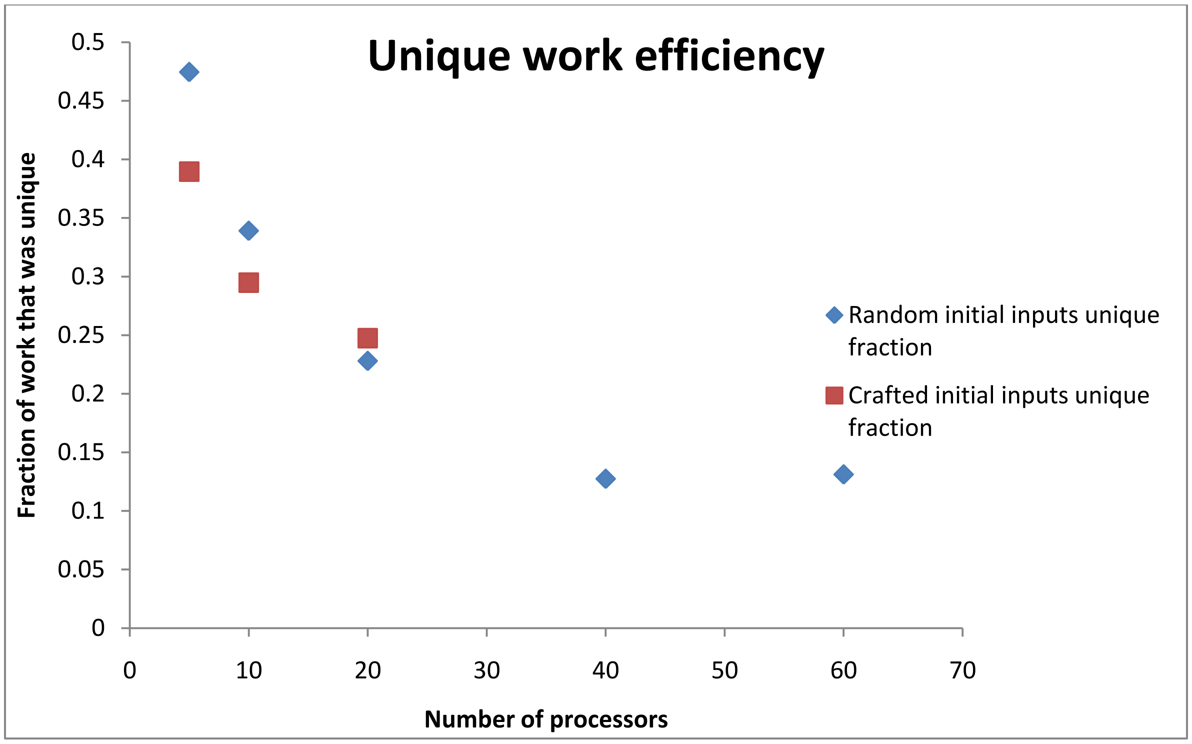


Figure 16: Unique work as a fraction of total work done by the random initial inputs and crafted initial inputs prototypes for test subject misc.003

prototype. In most cases, selecting inputs that explore well separated parts of the space of possible execution paths is not trivial. In fact, studies show that developers designing test cases believe they are achieving 90+% statement coverage, when on average they are actually achieving 50-60% statement coverage [87]. This becomes even more of a challenge when a developer tries to select test cases that not only cover different parts of the subject program, but that also are well separated in the exploration tree of concolic execution. In the generational exploration pattern chosen by GraCE, the analysis of each execution of the subject can produce multiple new inputs that that diverge from their parent at some branch point. However, what the child does after that divergence is not constrained in the SMT solution process. If two instances of GraCE that are acting independently to analyze the same subject, starting with different initial inputs, follow the same execution path through N branch points, and then diverge at branch point N+1, it is possible, and even quite likely that the SMT solutions produced in the different instances for inputs that behave differently at the first N branch points will be the same in both GraCE analyses[10]. This will prevent the exploration by independent agents from ever reaching 100% efficiency, no matter how the independent agents are seeded.

[10] It is possible and likely, but not certain. The analysis in the SMT solver is non-deterministic, so it may produce different solutions for the same set of conditions in any under-constrained case.

This subtle lesson in the nature of the concolic exploration of the space of all possible code paths was not obvious to us before experimenting with the parallel prototypes, but implies a limit to the utility of any independent agent design.

The shared queue prototype provided acceptably high efficiencies, as seen in Figure 17.

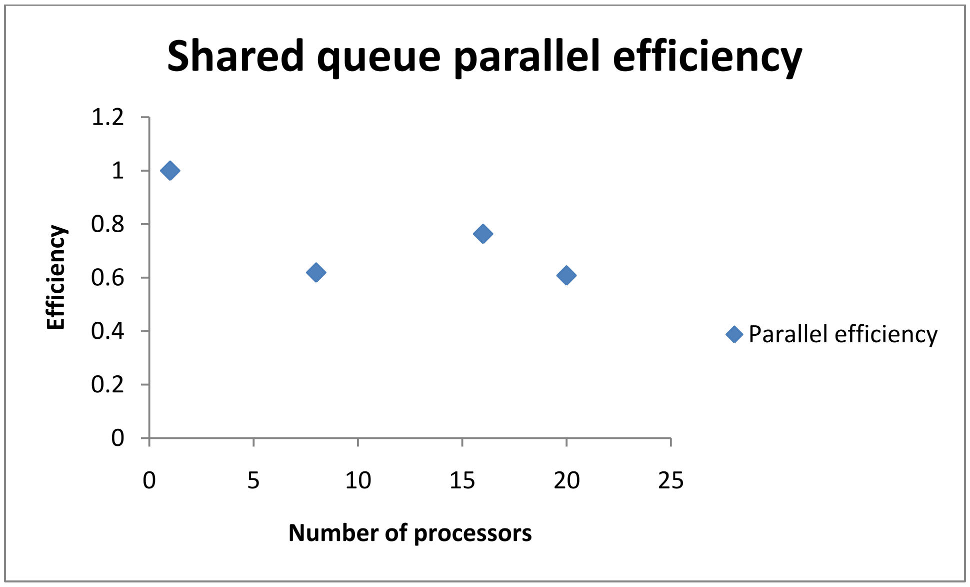


Figure 17: Parallel efficiency for the shared queue prototype running 5000 iterations on test misc.003

Again any projection to determine scaling is undependable. If there is a trend in the data, it is hidden by the noise in the measurements. A different approach is to look at the time the boss spends on communications for different cluster sizes, as seen in Figure 18. The scaling with extra processors in this case appears non-linear; extrapolating the communications costs indicates that communications costs will be more than 10% of run time with less than 40 processors.

The current prototypes only process command line inputs. In most programs, such inputs are very small, so the communications cost of transmitting them around the cluster is also small. However, the full set of inputs to a general program can be quite large. Images, spreadsheets, and substantial text documents are several orders of magnitude larger than typical command line inputs. Even though the current version of the core engine doesn't analyze such inputs, it is possible to simulate the cost of transmitting them. To do so, a small modification was made to the shared queue prototype so that each input description sent between the central queue and a worker was burdened with a character string of extra data[11]. The size of the string is a parameter that can be set when invoking the prototype.

[11] There is no need to consider extra input data for the random initial input and crafted initial input prototypes. They do not pass descriptions of the inputs between nodes during the analysis.

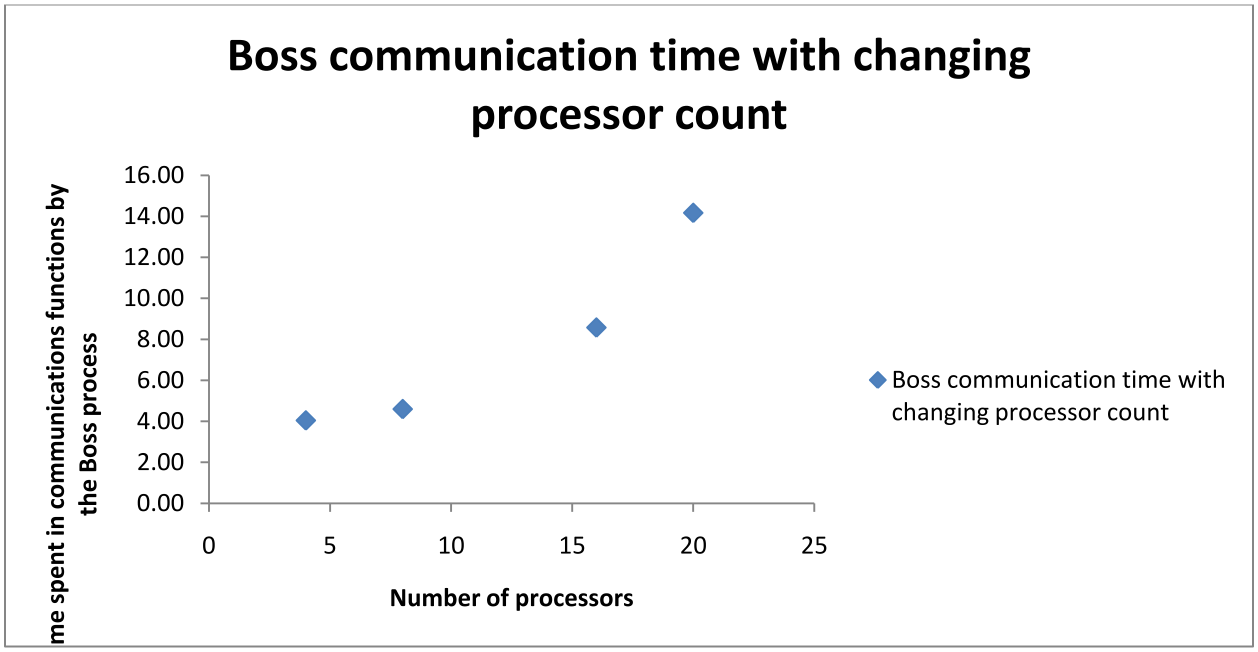


Figure 18: Boss communications times for 5000 iterations of misc.003 with the shared queue prototype. Total wall clock run time for 20 processors was ~800 seconds, so communications were never more than 2% of total run time.

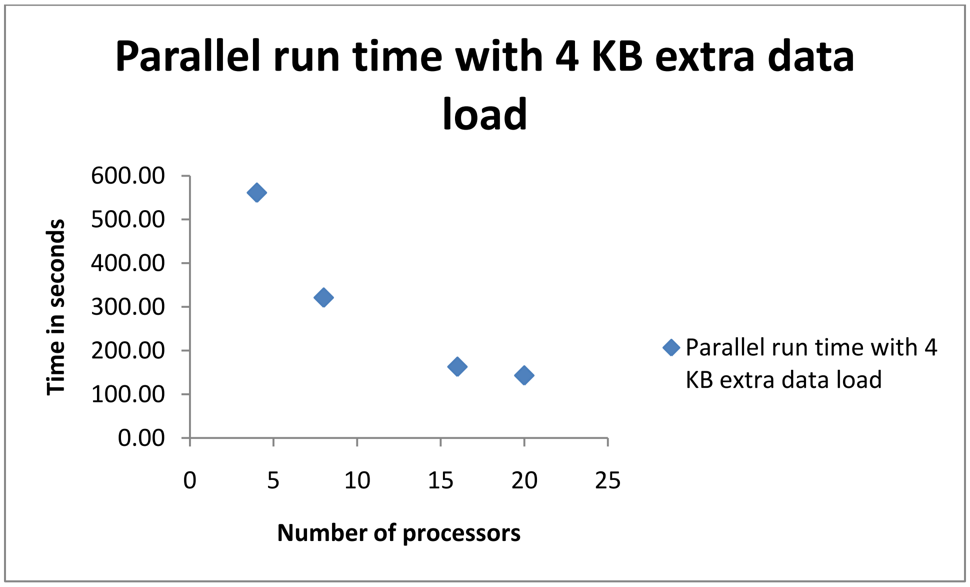


Figure 19: Wall time in seconds for 1000 iterations of misc.003 with 4 KB of extra data loaded on each communication of an input set. The times for this test are averaged from multiple runs of each configuration to minimize the influence of uncontrolled factors in the cluster. Run times are very similar to those measured with no additional load, and also show efficiency scaling close to the no additional load case

The first test for response to an additional load was to study the wall clock execution time for 1000 iterations of misc.003 with 4 KB of extra data. Typical generated inputs for this test are less than 100 bytes, so the data load was increased by a factor of more than 40 for this test.

Figure 19 shows that the scaling as the number of processors increases for this additional load is reasonable, while comparisons of the measured times with those for the no additional load case show very little change. 4 KB is apparently not enough load to affect the cluster operations.

The 20 node cluster has the most complex communications, so we would expect it to show effects from increased load before any others. It also has the shortest run time, and so is a good candidate for a test cluster for studying varying loads. For each value of the load, three data sets are collected and averaged, to reduce issues with uncontrolled variations in the conditions for the cluster.

The first interesting measurement is how the run time scales with the increasing load. Figure 20 shows that the scaling is approximately linear for a small load, and then changes character when the load passes some threshold. The graph is on a logarithmic scale, so change in slope implies a transition from scaling as one power of the load size to scaling as a different power of the load size[12]. The large increase in slope shows that the exponent in question has grown substantially. Generally such a change in character of the results happens when the dominant factor in determining behavior changes.

Figure 21 shows that the additional load has a substantial effect on the behavior of the boss process. In earlier examples the node running the boss process was underutilized. However, as the load grows, so does the utilization of the boss process node. For large data loads, the boss process is busy almost constantly. This makes the design change to place the boss process and a worker on the same computing node sharing resources far less appealing; programs with data loads on the order of 10-100MB will keep the node handling the shared queue busy.

[12] Remember that functions with power law scaling produce straight lines with slope equal to the power on logarithmic graphs.

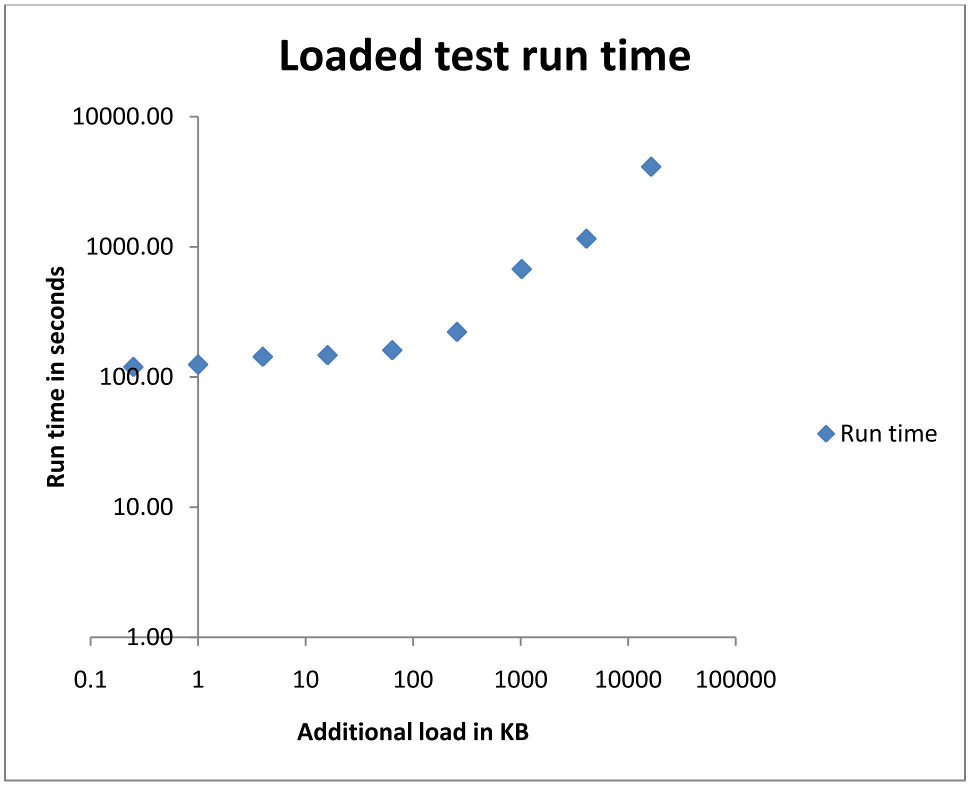


Figure 20: Time to run 1000 iterations of misc.003 on a 20 node cluster with varying extra data load. The load runs from a minimum value of 0.25 KB, up to 16 MB. Run times are reasonably stable with extra loads of up to 64 KB, but increase quickly with larger loads

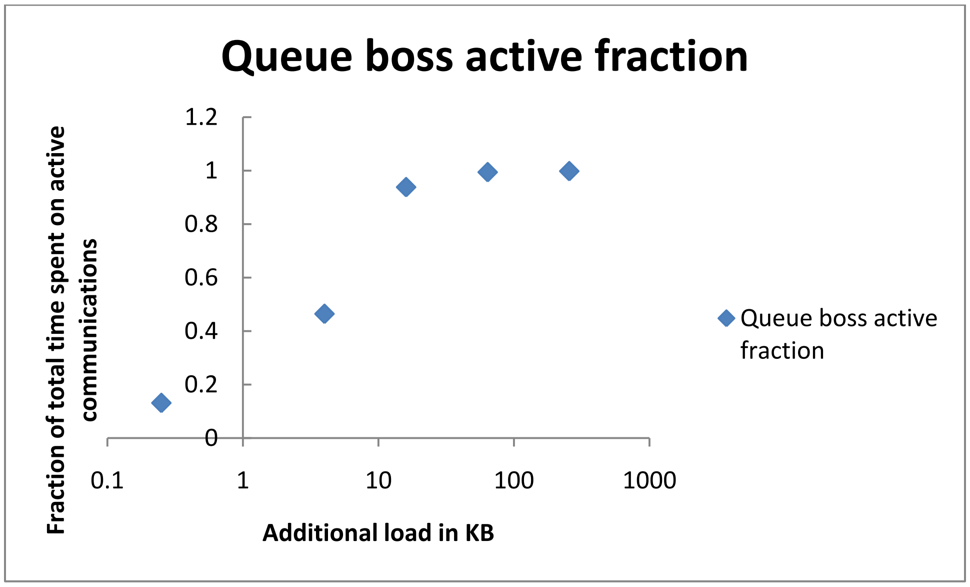


Figure 21: The fraction of the total run time for the queue boss process that was spent actively processing communications requests.

The worker processes are also strongly affected by the increased load. This is not hard to understand if we consider the way the extra load was mocked up in the prototype. No analysis is done of the extra data. It is transmitted and then ignored, only to be transmitted back at the end of the analysis iteration. So, the time spent on tracing the subject program and analyzing the trace is unaffected by transmitting the extra data. However, the run time grows by orders of magnitude. So, just as we see in Figure 22, we expect that the worker communication time will come to dominate the total run time for the workers.

This measurement shows a stair step like structure that is very reminiscent of a physical phase transition. It is beyond the scope of this investigation to determine whether there is a real phase transition in this behavior, and if so the nature of that transition. However, it is worth noting that similar phase transitions have been observed in small world network studies of information transfer. If this is a similar phenomena, we would expect the behavior to depend on the size of the cluster, the size of the data load, and how interconnected the communications are[13].

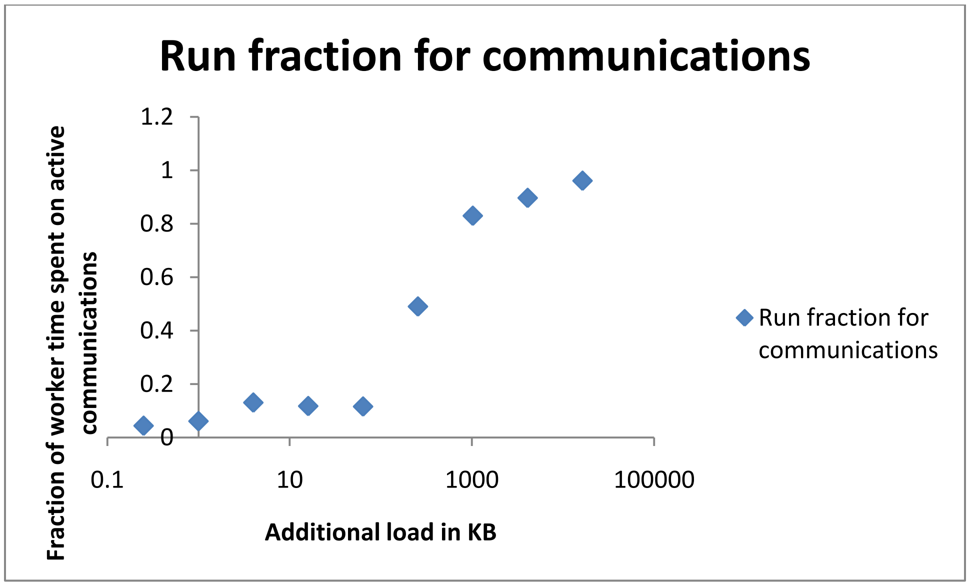


Figure 22: Fraction of worker process run time while analyzing misc.003 used in communications functions as a function of the additional data load. Note the extreme transition in behavior between 64 KB and 1 MB

The average time to send one megabyte of data decreases as the size of the individual messages increases. This is consistent with the design of MPI, where there is a fixed cost to establish a send/receive connection between two nodes, and then a per megabyte cost for the transmission. As we see in Figure 23, as the size of individual messages increases, this fixed cost becomes less important in measuring the communications bandwidth.

[13] The communications on an Ethernet network in a single subnet are very interdependent. Each node can see the packets sent by other nodes, and no two nodes can transmit packets at the same time. A more complex switching configuration would vary this constraint.

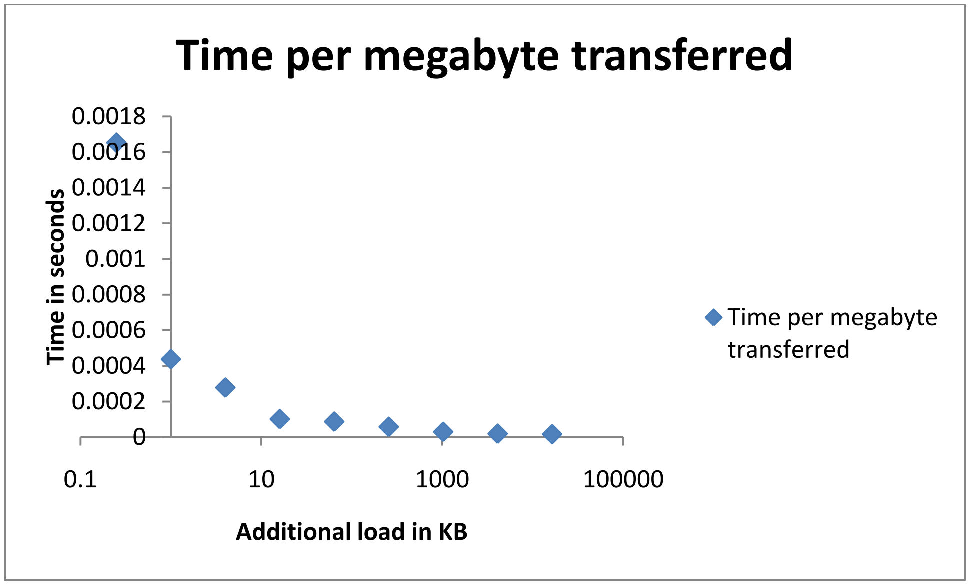


Figure 23: Average time spent transferring 1 MB of data between processes as a function of the additional data loaded into the process while analyzing misc.003. Transfers get markedly more efficient as the size of the data package increases

However, the messages still take longer as the size of the messages increases, as seen in Figure 24. This is again expected from the design, and is very consistent with a fixed connection cost plus a per byte time cost for sending messages.

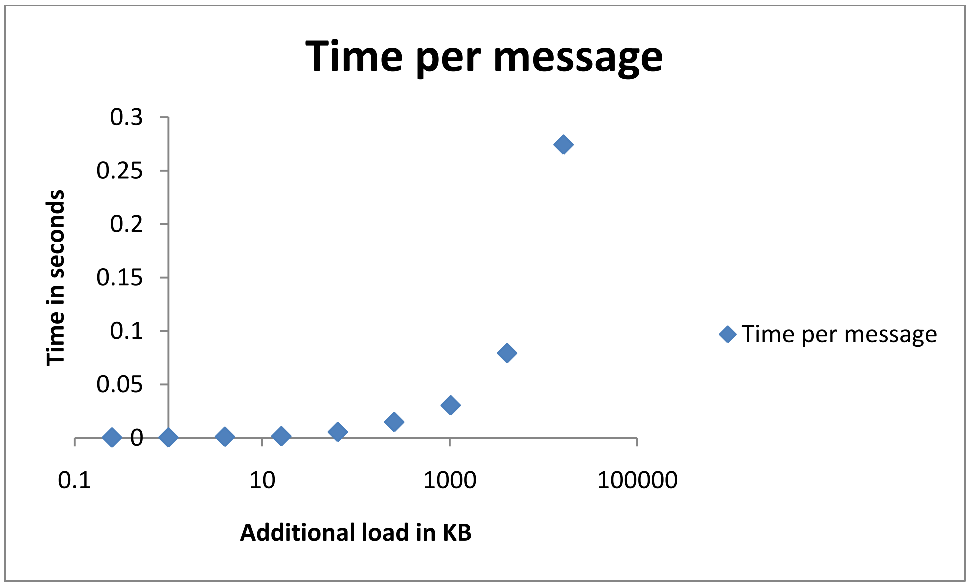


Figure 24: The average time taken to send one message containing an input set while analyzing misc.003, as a function of the additional data loaded into the input set. Although the figure above shows that the communications get more efficient as the messages get larger, this figure shows that the growth in size of the messages scales faster, so the net result is that each message takes longer.

The number of paths through the code is very large, so the path coverage for analyses with thousands of iterations is close to zero[14]. However, for all three prototypes the statement and branch coverage quickly reaches 100%.

- Misc.004: This program displays an overrun of a small buffer on the stack. Characters in the input are placed sequentially in the buffer, and if there are more than 20 characters, the buffer is overrun.
    - The program exposed a bug in the handling of register flags. The system exception handling code manipulates flags not normally accessed by user code. The flag handling code has been changed to understand these flags, and now also analyzes the exception handling code.
    - The exception handling code is extremely complex, and although GraCE analyzes it and produces inputs for further exploration without crashing, it is currently too slow to be usable.
    - Designs are under consideration to make this process more efficient.
    - This example is unusual in that the random-input prototype triggers the bug more quickly than the shared-queue design, because the random choice of inputs more quickly produces inputs that are large enough to trigger the bug. The path exploration for the shared queue tends to grow the size of the input one character at a time.

  Both the random and crafted seeds versions find the overrun – typically on the first iteration, and crash the subject and GraCE. The shared queue, that starts with argc=1 (no command-line arguments) as the first input, doesn't find the overrun in 10,000 iterations. The problem is that emphasizes breadth in coverage of the available characters and of the number of arguments as the first preference. Just adding new characters happens late, and every time it adds a new character, it checks that new character with many combinations of old characters. The longest input achieved was about10 characters, so the program would have run for a long time before finding the overrun. Lines of code coverage, and branch coverage were 100% for this, but path coverage was poor.

- No_validation.003: A program that does no input validation and searches the input for matching characters, reporting any matches.
    - The program doesn't validate its input, but the count of matching character pairs needs to see matching pairs to execute.
    - The random inputs design produces very few matching pairs, and so needs many iterations to explore the counting code. The crafted inputs explore that code right away. Thus the crafted initial inputs design has reasonable efficiency, but the random initial inputs design has poor efficiency, as seen in Figure 25.

[14] The code reads 4 characters and places them as the four bytes in a 32 bit unsigned integer. Then it uses a sequence of if tests to produce a second integer with the bits in the opposite order. This provides more than 2 to the 32 paths through the subject.

- Data for 20 processors with a shared queue prototype shows an efficiency ranging from 82% to 102% over multiple trials, so the scaling should be very good, and there is no chance to measure a potential scale issue from this. (The 102% efficiency is obviously connected to the changing and uncontrolled loads issue producing noise in the data. There is no theoretic reason that the design should be able to produce super-linear scaling)

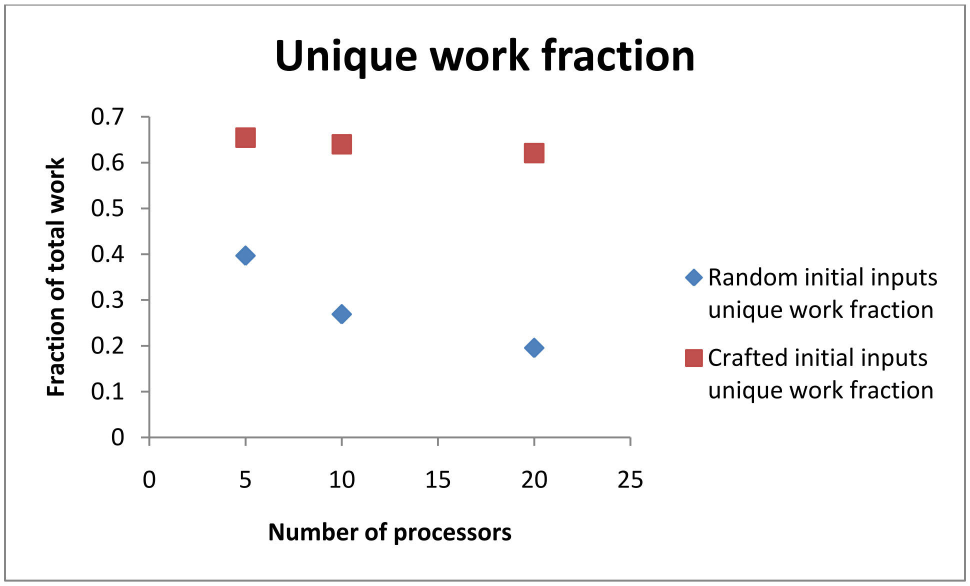


Figure 25: Fraction of unique work produced by the random initial inputs and crafted initial inputs prototypes for no_validation.003 as a function of the number of nodes in the cluster

	Serial	Random Inputs	Crafted Inputs	Shared Queue
Lines Executed	60.56%	95.77%	95.77%	95.77%
Branches Executed	68.00%	84.00%	84.00%	84.00%
Jumps Taken at Least Once	56.00%	80.00%	80.00%	80.00%

Detailed coverage measurements show that for 1000 iterations all three prototypes achieve high coverage. In a similar amount of time, the serial version of GraCE can perform 50 iterations, and coverage numbers are shown for that total.

- No_validation.005: This program looks for the first character in alphabetical order in the provided input.
  - This is a good example of a program where the efficiency for the crafted inputs is high, as seen in Figure 26.
  - Branch and statement coverage for 1000 iterations is close to 100%.

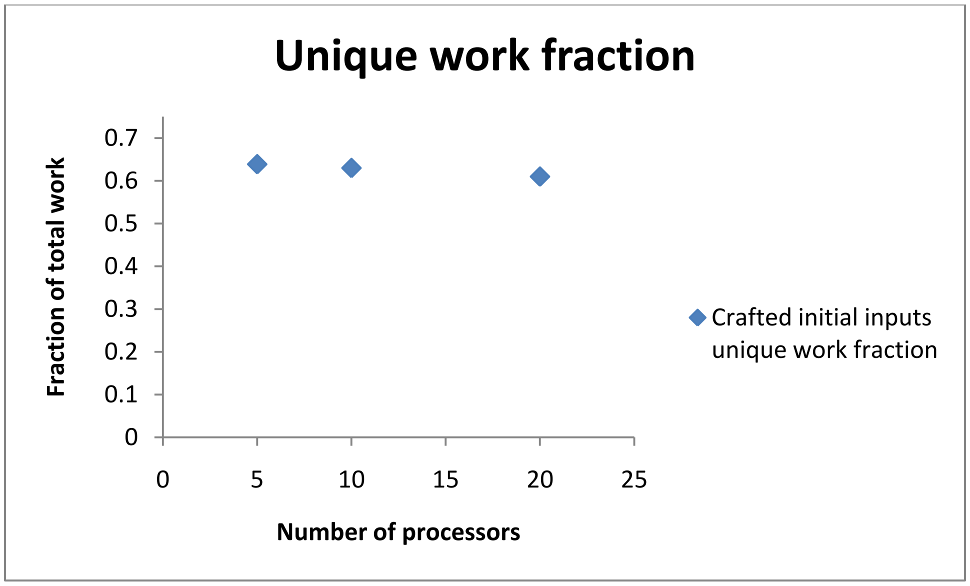


Figure 26: Fraction of unique work produced by the crafted initial inputs prototype for no_validation.005 as a function of the number of nodes in the cluster

- Validation.003: This program has substantial input validation (It chooses winning hands among input poker hands).
    - As expected the random-input design does exceptionally poorly.
    - The crafted-input design can do quite well, because the user can choose inputs that are known (because of knowledge of the program) to reach deep into the path space. See Figure 27.
    - The shared queue prototype with 20 processors produces efficiencies ranging from 67% to 89% based on multiple runs. The upper range of this data is quite close to the theoretic maximum of 95%.

Only the crafted inputs do well at exploring the code, however. The coverage for the other versions is very poor even for 1000 iterations. This is because the input validation is very stringent. The only inputs that succeed in validation are those for which the input matches possible poker hands in the chosen input notation. Only the crafted initial inputs prototype, with some provided inputs that provided valid hands managed to explore beyond validation in 1000 iterations.

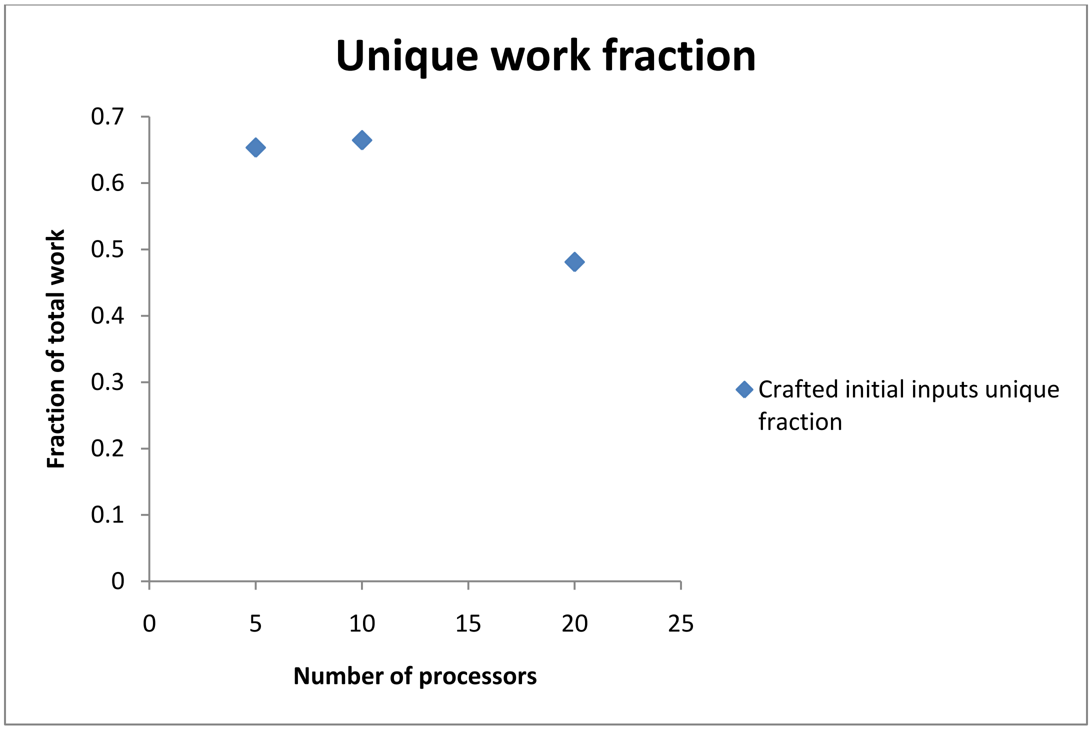


Figure 27: Fraction of unique work done by the crafted initial inputs prototype on validation.003 as a function of the number of nodes in the cluster.

	Serial	**Random Inputs**	**Crafted Inputs**	**Shared Queue**
Lines Executed	24.69%	24.69%	**92.89%**	25.94%
Branches Executed	23.64%	23.64%	**93.64%**	25.45%
Jumps Taken at Least Once	20.00%	20.00%	**86.36%**	22.73%

7.5.2 Medium-size programs that do real work.

These programs range up to a few thousand lines of code with compiled sizes that range up to .5MB. The execution paths are complex and sometimes non-deterministic. Thus, appropriate crafted inputs are not obvious. With these programs, analysis time (~90%) dominates subject program running time. Communication is less frequent and therefore communication overhead is not an issue (and is even difficult to measure accurately without introducing additional communication loads).

- Ai_test.002: This program does some simple natural language processing with a dictionary of about 60 words and several parts of speech and structures.
    - Shared queue communications time is less than 0.01% for 6000 iterations

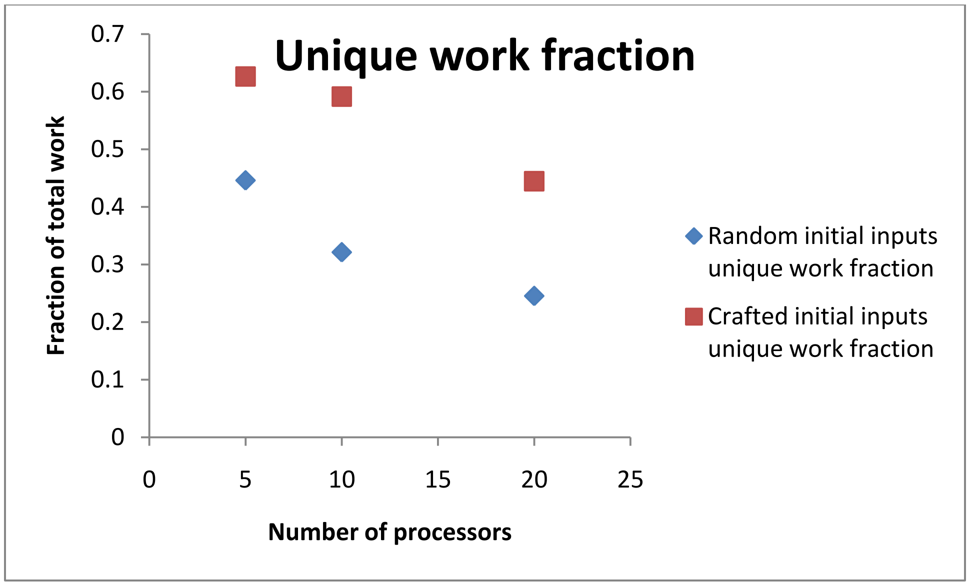


Figure 28 [AI_test.002] Fraction of unique work done by the random initial inputs and crafted initial inputs prototypes on ai_test.002 as a function of the number of nodes in the cluster

- No measurable trend in the communication time for the various cluster sizes is apparent for the measurements made so far.
- There is no way to project when the queue will become a problem from this data, since communications time is low across all our measurements.
- Queue behavior for large loads is measurable.
- Unique work for random seeds is better than expected, but still a poor ~25% for 20 processors.
- The crafted input design does better, but still has less than 50% efficiency for 20 processors.

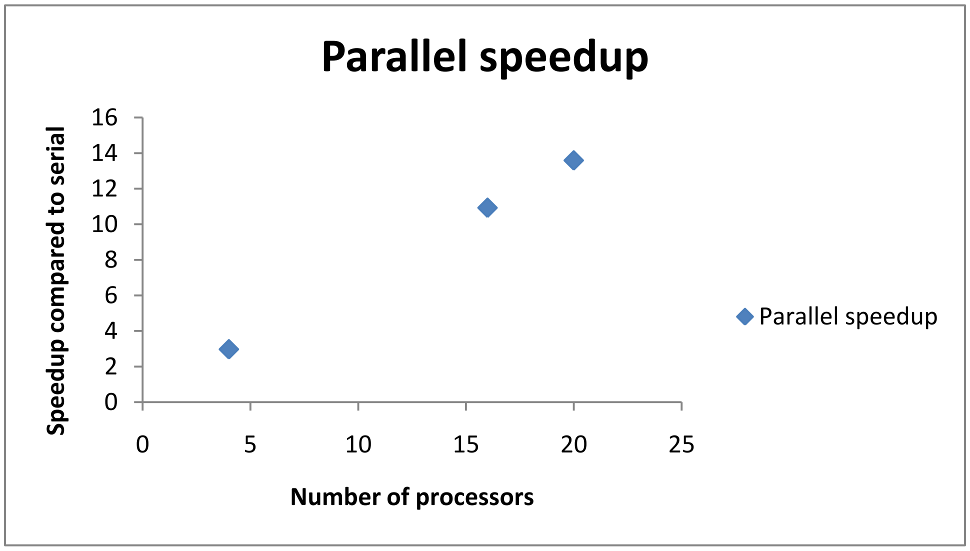


Figure 29: Parallel speedup for the shared queue prototype applied to ai_test.002

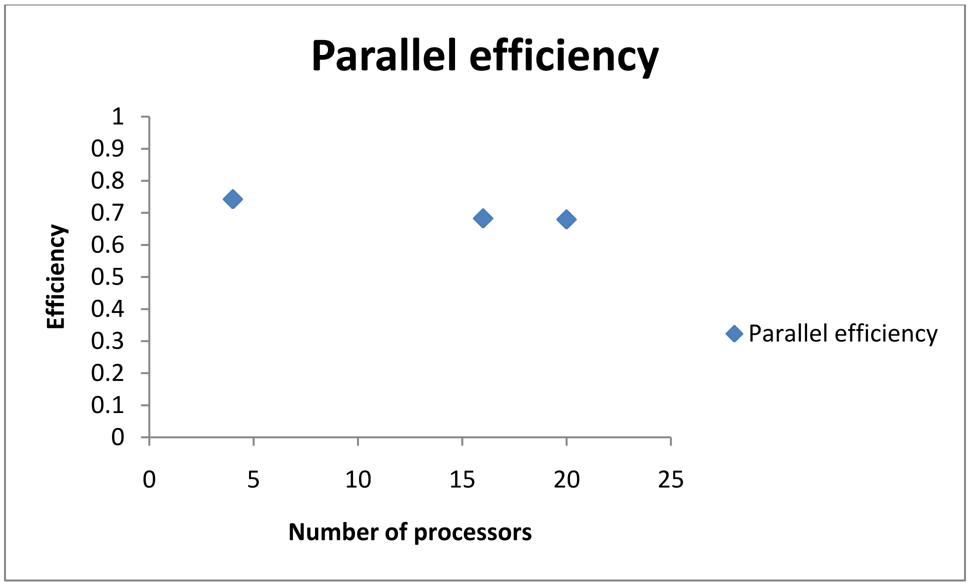


Figure 30: Efficiency for the shared queue prototype applied to ai_test.003 as a function of the number of nodes in the cluster

For the available cluster sizes, this class of larger programs scales well, with efficiencies close to 70%, and consistent to within the measurement errors induced by uncontrolled loads on the processors and network. Figure 29 and Figure 30 show the speedup and efficiency for the shared queue prototype.

We measured the start up costs associated with queue filling for this case, to provide some estimate of the effect of queue filling on the overall efficiency. The results indicate such costs exist, but they are inconclusive about how to project the costs to large numbers of nodes.

We ran several short trials with different numbers of processors and extrapolated the time spent on communications for a worker down to 0 iterations. If there are no startup costs, this would extrapolate to 0, but an offset in the intercept would show there is some initial cost before the processors are all running. Because the initial establishment of the communications in the cluster are not included in this timing, the cost is the time spent waiting while the queue fills enough to supply all the workers with tasks.

Figure 31 shows that the trends clearly do not extrapolate to zero, but they also don't give a clear estimate for the queue filling costs. (Two of the data points for 20 processors are hiding under points for 16 processors, and so are hard to see.) We did not include extrapolation lines in part because 2 of them have negative slopes, which is clearly unbelievable. The boss times for these runs (which would measure how long the actual communication took, without waiting in the queue) were less than a second, so the queue waiting time at the start of the run is substantial.

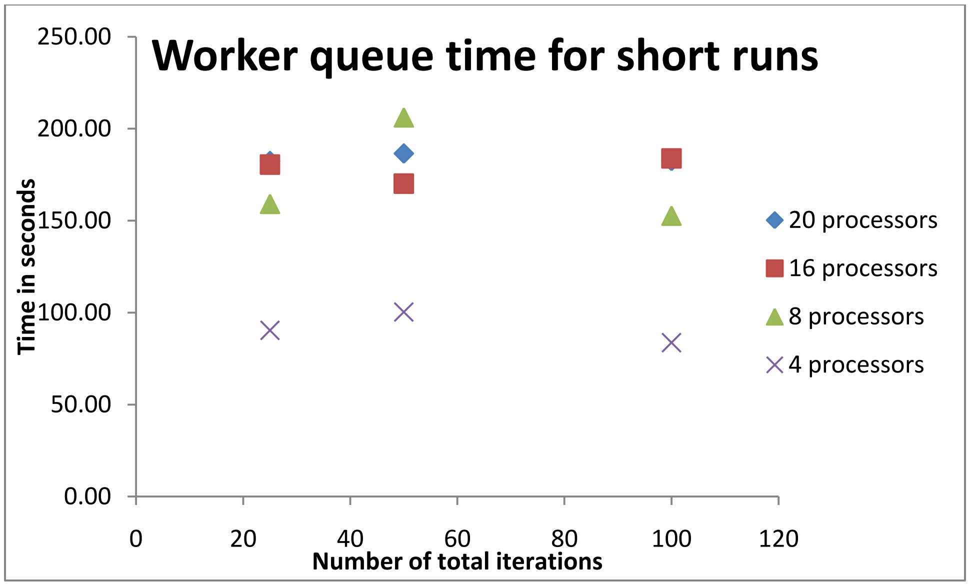


Figure 31: Measuring the start up costs due to queue filling for the shared queue prototype applied to ai_test.002. Note that the data values for 20 processors and 16 processors happen to overlay for both 25 iterations and 100 iterations.

A study of response to additional communications loads similar to that performed for misc.003 was also produced with ai_test.002. Similar results were seen, with an offset in the scale of the effect. This is because the individual runs of the subject and analysis are multiple orders of magnitude longer for this program than for misc.003, so communications effects of a similar time scale will be less noticeable in this setting.

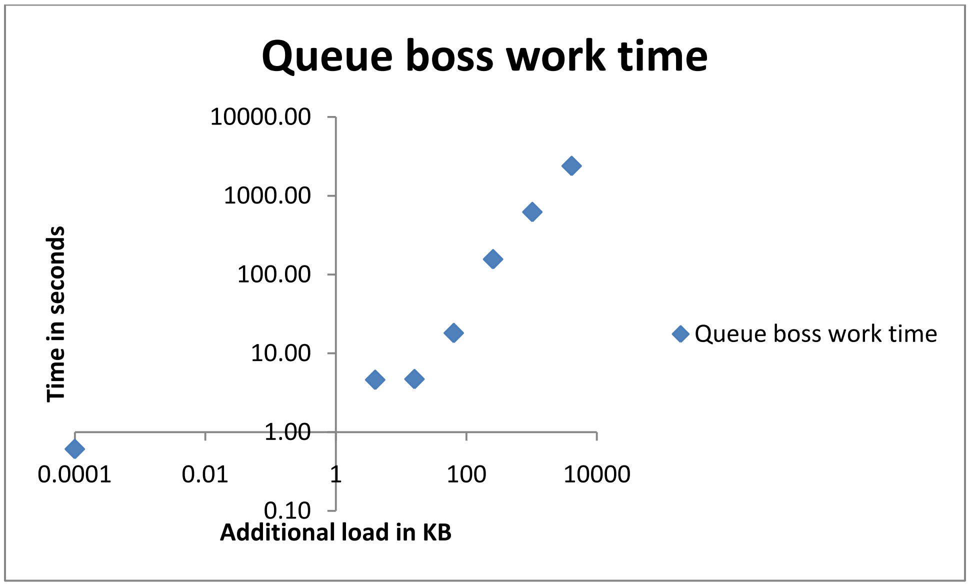


Figure 32: Total time spent performing active communications for the shared queue prototype as a function of the additional communications load.

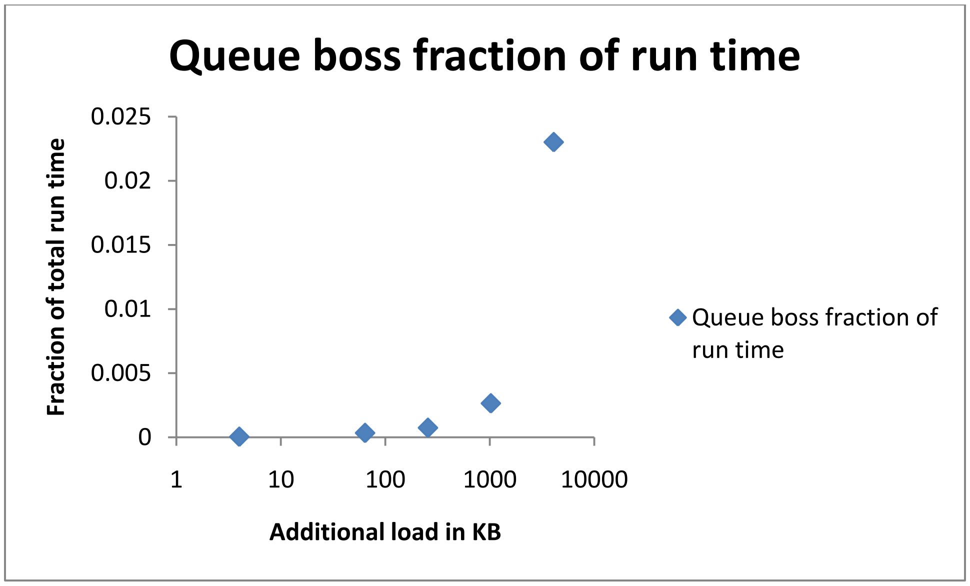


Figure 33: Fraction of the total boss process run time spent on communications for the shared queue prototype analyzing ai_test.002. Loads run from 4 KB to 4 MB, but notice even the worst case time is only just over 2%

What about traces produced by the random and crafted versions? The random version quickly drops below 30% efficiency, while the crafted version fares better as we see in Figure 28. This result may improve with a more insightful choice of initial inputs, but there is a real problem driving much of this drop off, as explained earlier.

Coverage numbers are poor for the serial engine; in the same time crafted inputs produce much better coverage using 20 processors. The measurement for the shared queue used twice the wall clock time that the serial engine used and 20 processors, but in that time produced almost complete coverage.

	Serial engine	Initial Crafted Inputs	2000 generated inputs shared queue
Lines Executed (of 168)	25.00%	88.10%	97.62%
Branches Executed (of 84)	7.14%	88.10%	100.00%
Jumps Taken at Least Once (of 84)	4.76%	67.86%	91.67%

- Ai_test.003 – an open-source ant colony optimization program. This program has non-deterministic paths because it uses randomization as part of the analysis.
  - The exploration found bugs in the code and caused crashes with crafted inputs and with shared queues where the iteration count was large.
  - However, with random inputs and decent iteration counts (~1600), the engine explored the initial validation code, which did not have crashing bugs.
  - Because all the random inputs were exploring validation code, the performance is quite poor.
  - The branch and statement coverage was about 25%.

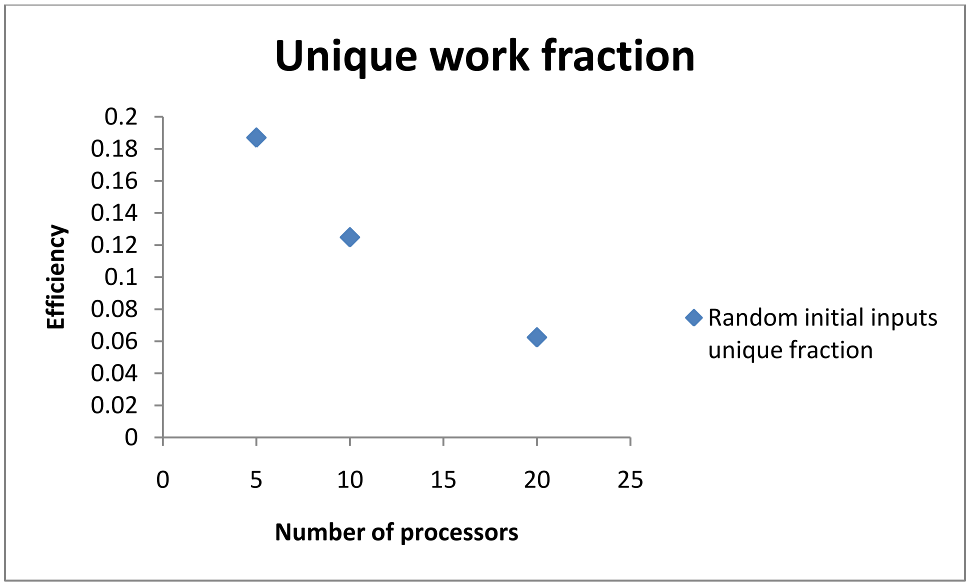


Figure 34: Fraction of unique work produced by the random initial inputs prototype applied to ai_test.003 as a function of the number of nodes in the cluster

- Ai_test.004: An A* search
  - This program requires some table look-up in the determination of paths, which the current engine does not support and so with future improvements we expect improved exploration.
  - There is substantial input validation that largely kills random exploration.
  - Some sets of 5 nodes in this data retrace the same collection of paths exactly
  - None of the random input workers escapes the validation section of the code
  - The shared queue prototype generates unique paths, but spends many iterations exploring the validation code without ever finding a set of inputs that enters the search code (of order 2000 iterations to find a valid input).

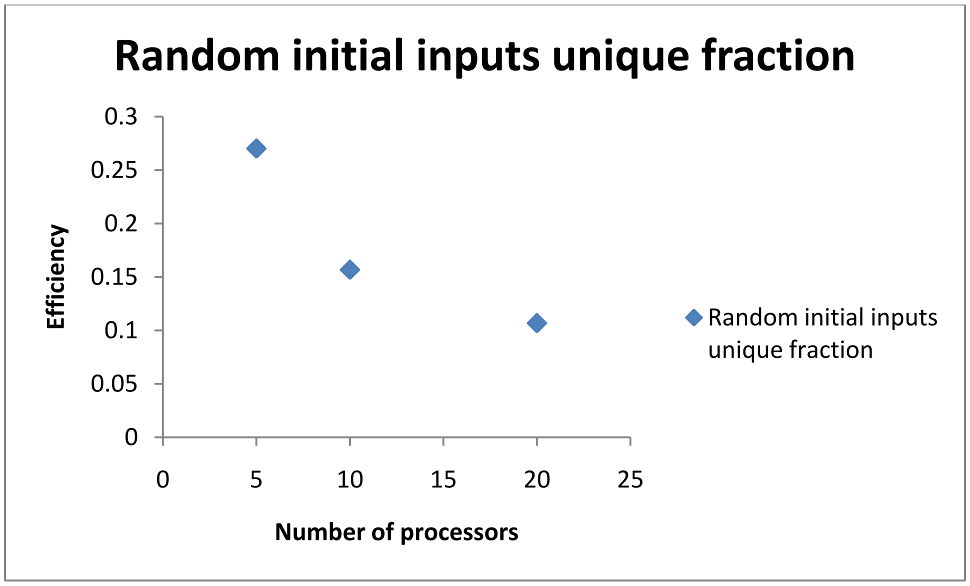


Figure 35 Fraction of unique work done by the random initial inputs prototype applied to ai_test.004 as a function of the number of nodes in the cluster

There is not a full study of scaling for the shared queue. Run times for the tests were long, so it was not practical to do many of them. However, for 20 processors, the efficiency was about 36%.

The number of new inputs produced by a single iteration was small (usually 1 or 2) according to the output logs, so queue filling was very slow. Worker queue time was more than 200 times as large as Boss queue time, which means that the workers were spending a lot of time blocked waiting for new input. Boss queue time was small compared to subject run time and analysis time, so the Boss had very little communications work to do. This is exactly what you would expect when the queue is empty for much of the run time, and the workers are idle waiting for something to do.

This gives an example of a type of program that is hard for parallel concolic to analyze: one in which each new iteration sees very few unexplored branch points. In this program, that is because many of the branches are determined by table look-ups. As we enable GraCE to explore table look ups intelligently, results on this test will get much deeper, very quickly.

7.5.3 Larger programs that do real work.

These programs are large and complex enough to be the main body of a real application. They have been modified to not use file input (all input is as text on the command-line) because of some current limitations of GraCE. The analysis times are very long; analysis in serial mode is impractical. The code sizes are 10K to 100K source lines and multiple MBs of compiled code. We have no fully analyzed examples in this category. They were useful as demonstrations of capability (and of issues needing repair), but the running time was too long to produce full analyses or repeated experiments within the time-frame of this project.

8 Reliability and performance testing of the concolic engine

A key part of developing a new capability is testing it for consistency and reliability during the development process and for regressions after primary development. In the work under this heading, we put in place a testing framework, built on the regression testing facility that GrammaTech uses for other research and commercial projects. In addition, the framework was extended to measure the performance of various parallelization options.

Sequential Concolic Test Script: The nightly regression testing script includes checks to thoroughly determine successful concolic execution of a subject program. Correctness checks are performed for each test case – including recognition of basic concolic errors in the execution logs and checks of the full log output. The test script also allows the tester to specify initial inputs to the concolic execution. If desired, the tester can also provide critical inputs that a concolic execution is expected to generate, and the script provides the ability to make the existence of these inputs a success criterion.

The infrastructure allows us to

- Easily add new tests to a test suite, with given expected outputs
- Categorize tests as ‘expect to succeed’ or ‘expect to fail’. The latter category might include test cases that exercise capabilities that are not yet implemented or have known bugs.
- Run the test suite as part of a nightly buildbot process, with the results available on a web page as the tests run.
- Run the test suite on a variety of machines, either as serial processes or as groups of parallel cooperating processes.
- Use the infrastructure to measure both raw execution performance and code coverage.

The infrastructure is useful in maintaining a working tool within this project but will also be useful in follow-on projects that use the GraCE tool.

With the infrastructure in place, the test suite was greatly enhanced. It now includes groups of tests to assess the following:

- Tests that check that individual machine code instructions are being properly interpreted by GraCE.
- Regression tests that include examples that were problematic at one point in the past, are now fixed, but continue to be tested so that they remain fixed.
- Tests that exercise the parallel processing capability.
- Tests that use real-world, open-source programs adapted where necessary to run with GraCE. These are larger scale, longer-running programs.

Coverage Analysis: The GraCE tool is intended to find new execution paths in a subject program. If the program has only a finite number of paths, the tool should find all of them. If the subject program has an infinite number of paths, the tool’s search algorithm should concentrate on the most important. Thus it is useful to have a metric by which to measure the quality of set of execution paths that GraCE finds.

As an initial measurement of GraCE's performance, we use line and branch coverage metrics. That is, we use GraCE to generate a set of inputs that execute a subset of the subject program, then use those inputs as a test set for the subject program, and then measure the line and branch coverage we obtain with that set of inputs. Line coverage is the percentage of lines in the source code that are touched by at least one execution of the subject program, while branch coverage is the percentage of branches covered by at least one execution of the subject program. Branch and statement coverage can sometimes be misleading, because simply having even complete branch and statement coverage does not mean that all interesting combinations of behaviors have been tested. Nevertheless, the metrics do provide some initial feedback as to how a parallel prototype is performing relative to the baseline sequential execution on the test suite.

Parallel Concolic Test Script: The test script allows multiple versions of the concolic engine to be executed on each test case in the suite. For each parallel prototype created, nightly tests can be executed to verify correctness and report performance gains against the baseline using the aforementioned coverage metrics, as well as timing metrics. On a single test case, we can obtain line and branch coverage values for the baseline sequential concolic execution and for all parallel prototypes. These results can then be compared to identify performance changes experienced by the various implementations. Many of the future parallel performance measurements will branch from the base laid out here.

Subject program error detection: An area of testing commonly missing from test suites is error recovery. In GraCE's case, when the subject program fails, we want to be sure that the controlling engine does not itself fail. Rather, when a subject program reaches an instruction that produces an error (such as a divide by zero or invalid memory access) the concolic engine should be able to detect this and handle the error gracefully. Such tests are particularly crucial for GraCE since a goal of concolic execution is to find very rarely executed code branches which are then more likely to be buggy or more likely to be malicious. Tests were added to ensure that error paths were tested.

9 Robustness and scalability improvements to the concolic engine

As part of using and testing GraCE as a component of a broadly distributed tool for finding malicious code, we needed to improve the robustness and scalability of the core concolic engine. We concentrated our improvements to the concolic engine itself in the following areas:

1. Validation of the underlying semantics
2. Handling of alternate input mechanisms
3. Correction of problems that inhibit scalability of the engine

Item (1) is discussed in section 10.

Item (2) requires a change in the QFBV logic library that encodes the semantics. This change is completed, but not the additional implementation of alternate input mechanisms. While it would enlarge the set of real-world programs that could be used in assessing parallelization, it is not essential to the fundamental results of the intended research program.

Under item (3) we investigated a number of problems, as described below.

9.1 Optimizing concolic execution

9.1.1 Modify and improve SDT utility

Concolic execution depends on software dynamic translation (SDT) to produce the traces and context information used in the analysis. The utility used by GraCE is an in-house SDT tool, so the source code is available to the developers. Inspections of the code and its use showed that there were several instances where system-wide resources were used in a way that was not compatible with concurrent operations. Consequently, the utility code was modified to improve the ability to run in a concurrent setting.

This work included removing the use of some environment variables that enabled interprocess communications and changing the invocation method for the tool so that it is a statically linked library instead of a separately invoked executable.

Several other smaller issues, and some interface clean up were addressed at the same time.

9.1.2 Modifying the concolic engine

GraCE wasn’t originally developed with concurrent operation in mind, so it also had some potential issues that had to be addressed. In addition, the changes in the SDT tool also implied changes in how SDT is invoked in the concolic engine. Also, there were some simple interface changes that were needed to support some of the prototypes.

Specifically, the engine was modified to accept arbitrary initial inputs for the subject program from the user, instead of generating the initial input internally. The use of environment variables to communicate with the subject program was removed; the use of named pipes for more extensive communications was modified so interactions between concurrently running controllers are eliminated. Some improvements to the base serial engine were produced, extending its range of utility and its stability with regard to a failing subject program.

The use of the concolic engine was also modified to separate the functionality from the calling methods. Consequently, the engine can be used in some of the prototypes without need for any large scale changes in the code.

9.1.3 Installing and testing the framework for distributed computation

One of the major complications of distributed computation is the overhead of starting processes, controlling their behavior, and passing messages. This overhead is largely abstracted away by the use of the MPI framework as part of the distributed code.

MPI is a separately specified distributed computing framework built for cluster computing with very low additional footprint and high speed communications. It is a very common tool in the High Performance Computing community, and it is well suited to problems where the number and configuration of nodes in a cluster is known in advance. It is also fairly easy to transform a cluster implementation into a grid implementation, since many grid computing projects began as cluster projects written using MPI.

The implementation of MPI we chose to use is MPICH2, from Argonne National Laboratories. It is widely used, efficient, stable, and works well in a Windows XP environment. It implements the MPI 2.2 standard, and it provides a wide palette of methods for control and communications.

MPICH2 was installed and tested on multiple local test machines. It worked correctly and supported prototype development.

9.2 General problem correction

Handling subject program crashes.

A key improvement was to make GraCE robust against crashes in the subject program. Since GraCE intentionally looks for problematic subject program code, this improvement was an important one to make. Finding problems of this sort was a key goal of the effort to build and extend the test framework described in Section 8.

Failures in Windows pipe connections. One issue highlights a problem with the overall Windows environment. The GraCE concolic engine runs and monitors the subject program using two components: (a) an execution engine that controls the overall execution and (b) the subject program itself, to which we attach a software dynamic translation module (SDT) that provides (among other services) communication back to the execution engine. That communication happens through Windows pipes; these pipes are reconstructed for each run of the subject program. The construction of Windows pipes can randomly fail (Windows does not guarantee that the operation succeeds). The failure rate is low, but, over many iterations, the number of failures is noticeable. The failure happened often enough to affect long running data collection. This problem was analyzed and corrected. We replaced the named Windows pipe with a cross platform shared memory resource provided by Boost Interprocess mechanisms. In addition we changed the design to construct the pipes only once (rather than on each iteration), further reducing the failure rate.

The increased reliability will be a benefit for the usefulness of a distributed system overall.

Stack Overflow. Part of our test suite exposed GraCE to inputs that are strenuous for the engine. These stress tests showed that the engine overflows its stack space when it encounters basic blocks of very large size. Because basic blocks of the size required for crashing GraCE are not usually encountered in realistic programs, we chose to address this issue by increasing the size of the stack space allocated for the concolic-execution engine rather than by proper elimination of recursion. Increased stack space also addressed the issue of stack overflows in the Yices SMT decision procedure that we encountered in our previous work on the engine. Yices, developed at SRI International, is an essential component of the concolic-execution engine—it is used by the engine to derive program inputs from logical encodings of execution traces. While we have the source-code license for Yices, we would prefer to avoid modifying it to eliminate stack overflows.

Division. We also identified and fixed bugs in our handling of bit-vector division operations. The version of Yices we are using does not provide bit-vector division as a native operation. Thus, we modeled division by introducing two free logical variables—one for the quotient (Q) and one for the remainder (R)—and adding corresponding constraints on the values of their variables. Let A denote the dividend and B denote the divisor. Our implementation of division follows signed integer division from the C programming language. Specifically, we add the following constraints:

$$A = Q \times B + R$$

$$0 \leq |R| < |B|$$

$$A < 0 \Rightarrow R \leq 0$$

$$A \geq 0 \Rightarrow R \geq 0$$

The first constraint links together the values of A, B, Q, and R. The second constraint makes sure that the remainder does not exceed the divisor. The last two constraints make sure that the remainder gets the proper sign (the same as the sign of the dividend).

The formulation of the logical constraints can affect the efficiency with which solvers such as Yices can solve systems of constraints. For example, the last constraint can be split into two pieces:

$$A > 0 \Rightarrow R \geq 0$$

$$A = 0 \Rightarrow R = 0 \textbf{ and } Q = 0$$

This more closely constrains the logical system, which usually improves the performance of Yices, but it also adds more statements to the system, which usually degrades the performance of Yices. Experiments showed that the formulation above is about 30% faster.

Our original implementation of the division operation failed to account for the possibility of overflow in the multiplication $Q \times B$. In one of our division tests, Yices picked the value of Q to be large enough to cause overflow, which prevented the concolic-execution engine from generating sufficient coverage of the program. We diagnosed the problem and fixed it by increasing (doubling) the width of all the bit-vectors involved in the above constraints—this eliminates the possibility of overflow and corrects the problem we encountered.

Cleanup. Also, we significantly cleaned up the code of the concolic-execution engine. Original development of the engine explored several possible options for implementing concolic-execution, such as using different methods for encoding execution traces and different search strategies for generating inputs. Most of these have become obsolete, but were still cluttering the source code. We have eliminated the obsolete components and simplified the concolic-engine infrastructure wherever possible.

A portion of the concolic-execution engine had been developed jointly with a team from Advanced Physics Laboratory at John Hopkins University (APL-JHU). APL was our subcontractor in the OSD/AFRL-sponsored STTR under which GraCE was originally developed. During this reporting period, we have revisited that code, extended it, and moved it into GrammaTech source-code repository.

Assessment of correctness and scalability. Extensive thought and study was dedicated to finding possible limitations or incorrect behaviors of GraCE and diagnosing them. The goal in this work was to isolate root causes and improve the efficiency, stability, and scalability of the concolic engine.

A number of technical work items were identified, and plans were developed to address those items. Concerns include the model for errors and interrupts at the processor level, stability under unexpected behaviors of the subject application, improvements to the generation and simplification of SMT queries to increase solution quality and reduce processing time, reorganization of trace collection communications to reduce the need for expensive two way communications, and other items.

10 Correctness of the underlying machine code semantics

A fundamental aspect of all concolic execution, including GraCE, is the translation of the program into a logical representation. The logical representation is used by a constraint solver to reason about the various program paths. Each concolic engine must implement, implicitly or explicitly, such a translation from program to logic for each programming language that it interprets. GraCE translates machine code instructions. If the underlying translation is inaccurate, the concolic engine will malfunction.

In this portion of the project we conducted experiments to determine the accuracy of the translation that is used in GraCE. That translation uses a language called TSL (Transformer Specification Language) [59] to represent a mapping from each machine instruction to a statement in a chosen logic, in this case, Quantifier-Free Bit Vector logic – QFBV. In this section of our final report we describe TSL, the mechanisms used to test the accuracy of the QFBV translation, and an extension that tests general abstract interpretations.

10.1 TSL

GraCE uses the Transformer Specification Language (TSL) along with a description of the processor instruction set architecture (ISA) and an analysis target to translate the semantics of the processor into a logical representation. Given a TSL specification of a particular ISA, such as x86, TSL generates a logical representation of instructions in that ISA that can be used for analyzing executables written in that ISA. It does so by instantiating the TSL specification with an *interpretation* that specifies the particular kind of logical representation desired. One advantage of using TSL as a specification language for processor semantics is that if one has *n* TSL specifications for various ISAs, and *m* interpretations for various representation techniques, then one can generate *n*m* analysis engines in much less time than it would take to create them by hand by separately writing the *n* ISA specifications and the *m* interpretations. Thus, *n*+*m* effort produces *n***m* results.

10.2 TSL Validation

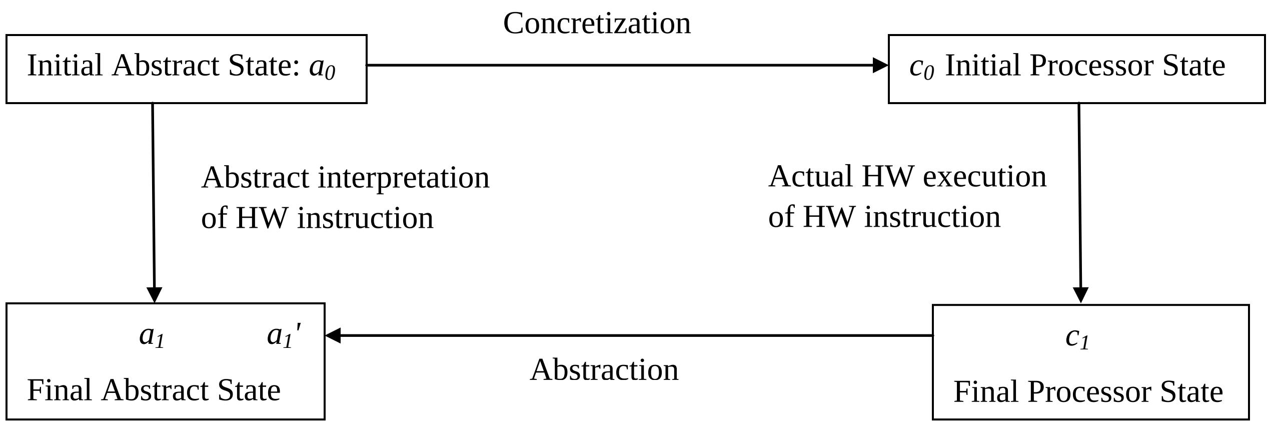
To create a TSL validator for a particular ISA, we first instantiate its TSL specification with an interpretation that defines a software emulation of the processor, thereby generating a CPU emulator. The software emulation interpretation should maintain a state that is directly translatable into the state of the processor, so the changes in state for the emulator and the physical processor can be compared for different instructions in the ISA. (Such an interpretation was already implemented and provided to the Safety In Numbers project; the point of this exercise was to validate it.) We can then compare the generated emulator's behavior to that of the physical processor for a number of different instruction streams and initial processor states. If running the same test on both produces different results, the TSL specification is likely to be wrong.

10.2.1 EmuFuzzer

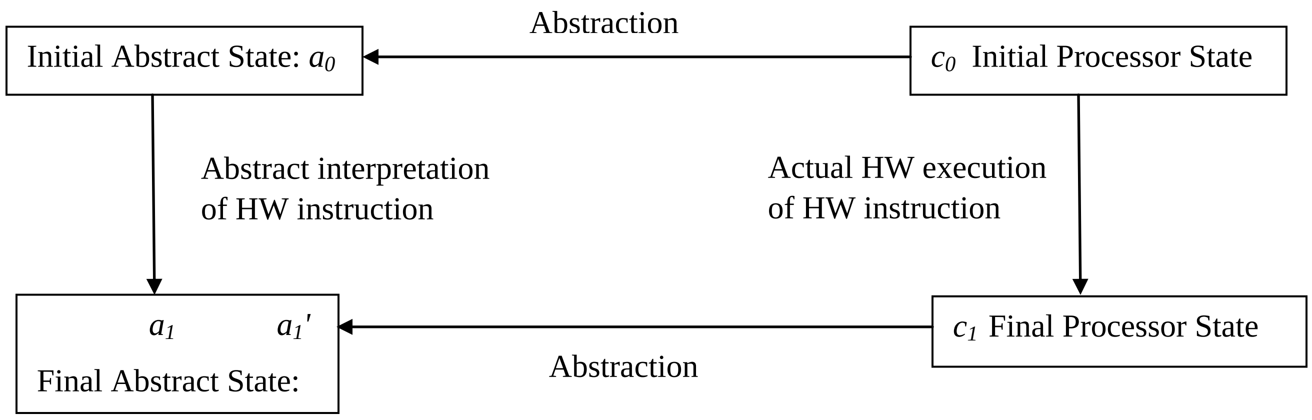
For the purpose of comparing the behavior of the emulator and the physical CPU, we considered the use of a third-party tool, EmuFuzzer (cf. http://martignlo.greyhats.it/papers/EmuFuzzer.pdf) for testing CPU emulators. Under EmuFuzzer, comparison of the emulator and the physical processor proceeds roughly as follows.

Begin with an abstract state, a_0, representing the memory and registers of the emulator before any code is run. Executing an instruction in the emulated environment produces a new abstract state a_1. In the case of an emulator, an "abstract" state should not be very abstract at all; it should capture the entire state of the emulated processor. However, we also want to use this tool for interpretations that transform the instructions into logical representations where the mapping between the state of the processor and the abstract state is more abstract and less precise. In such a case, the abstract state describing an integer register might be "*the contents are odd*," or "*the contents are negative*.")

Meanwhile, EmuFuzzer applies a *concretization function* to the original abstract state a_0 to produce a concrete state c_0 representing the memory and registers of the physical machine. Executing an instruction in the physical environment produces a new concrete state c_1. Finally, apply an *abstraction function* to c_1, resulting in an abstract state a_1'. For any interpretation, we expect $a_1 \supseteq a_1'$ to hold. For the emulator interpretation, we expect the stronger condition $a_1 = a_1'$ to hold.


Concretization
Initial Abstract State: a_0
c_0 Initial Processor State
Abstract interpretation of HW instruction
Actual HW execution of HW instruction
a_1 a_1'
Final Abstract State
Abstraction
c_1
Final Processor State

However, an alternate and simpler design allows the same comparison to be made. In the alternate design, we start with a concrete state c_0 and apply the abstraction function to it to produce an abstract state a_0. We can then execute an instruction in the emulated environment to produce a_1; execute the instruction in the physical environment in state c_0 to get c_1, and apply the abstraction function to c_1 to produce a_1'. Notably, this design eliminates the need for the concretization function.


Abstraction
Initial Abstract State: a_0
c_0 Initial Processor State
Abstract interpretation of HW instruction
Actual HW execution of HW instruction
a_1 a_1'
Final Abstract State:
c_1 Final Processor State
Abstraction

EmuFuzzer is itself a new research project and not well documented, so it was unclear whether its design would be suitable for our needs until we looked at the code. Our investigation has resulted in a decision not to use EmuFuzzer in the TSL validator, for the following reasons:

- We would like our design for the validator to use a behavior we call *look-through semantics*: when an abstract state is needed in the emulated environment and necessary aspects of the state are missing, we "look through" to the physical environment and use the representation given there, then perform an "on-demand" application of the abstraction function to get the abstract state we require. In EmuFuzzer's design, the look-through behavior is in the wrong direction—from the physical environment to the emulated environment, rather than from emulated to physical—and the function performed on-demand is concretization, rather than abstraction.
- EmuFuzzer's test framework requires the generation of "test cases" specifically for testing purposes. Although this design can allow for good test coverage, it does not support attaching to an arbitrary running process and testing the instructions being executed in that process.

In principle, these issues could be overcome, but taken together, they make working with EmuFuzzer prohibitively difficult. Moreover, we would eventually like to use the validator not only for validating TSL specifications but also for validating interpretations, and the on-demand concretization-based EmuFuzzer architecture is not well suited for such a purpose, because implementing the concretization function is generally much harder than implementing the abstraction function. We have therefore opted to design our own validator architecture rather than one based on EmuFuzzer.

10.2.2 GrammaTech's validator

Our design specifies that the validator will operate on test programs which may be arbitrary executables. It will have, at a minimum, the following capabilities:

- Some standard debugging capabilities, in particular the ability to single-step a test program one instruction at a time.

- The ability to set write-protection (and possibly read protection) on pages in a test program's memory space.
- The ability to handle write protection (and possibly read protection) faults from a test program.
- The ability to emulate instructions of a test program, using look-through semantics.

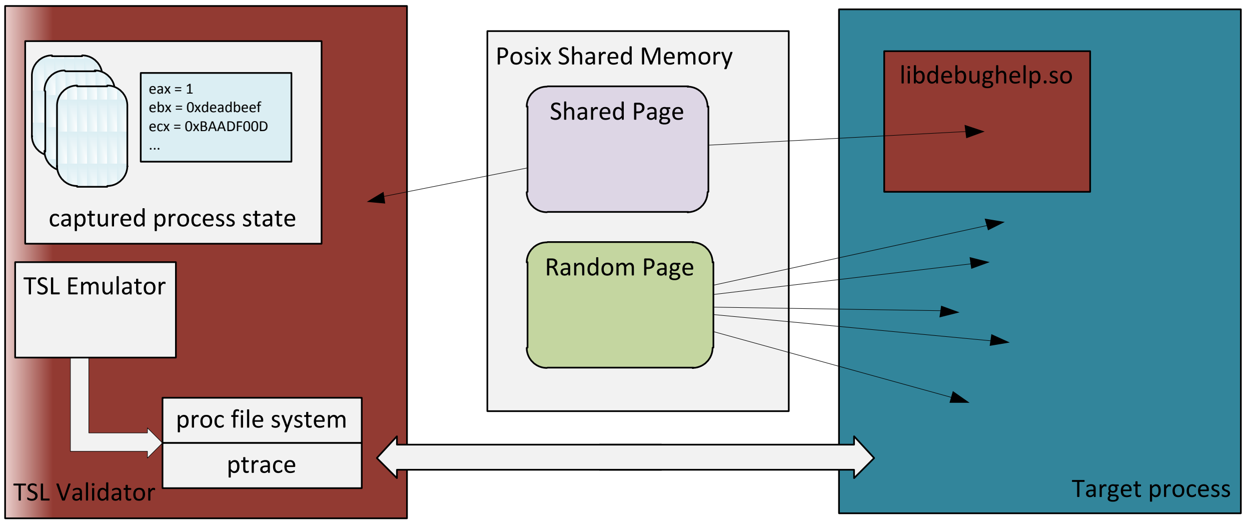
Finally, since it is infeasible to exercise all the possible behaviors of the processor in our test programs, an important aspect of the validator will be choosing inputs to the emulator that will be most likely to reveal bugs in the original TSL specification. We are interested both in testing arbitrary programs to expose bugs in the specification that would be realistically encountered, and in broad-coverage testing of individual instructions. Our separate work on concolic testing may be of use for the latter.

An important aspect of our validator is that it is capable of directly checking the actual machine code program against an interpretation of the program – it is not an emulation of software. As we implemented it, the validator uses a memory protection scheme and the debugger API to single step through a process, capture its state modifications, and then compare the results obtained on the actual machine to those obtained by the emulator. This allows a TSL definition of "concrete CPU semantics" to be checked directly against a real CPU, at least for the particular sets of input memory and register states that are tested.

Furthermore, by exploiting the mathematical properties of "Galois Connections", the validator also provides a mechanism for validating abstract interpretations. By capturing the concrete state of executing a real program on a real CPU, and applying an "alpha function" to map its physical state into an abstract domain, the validator provides a simple, effective, and cheap mechanism for verifying the soundness of an analysis implementation. Finally, the validator supports two modes of operation:

1. "Normal" mode, which allows a TSL specification to be tested against arbitrary, real world programs.
2. "Fuzz mode", which supports fuzz testing (by executing random instructions over random data).

10.2.3 Validator Architecture



The TSL Validator is composed of 3 main components:

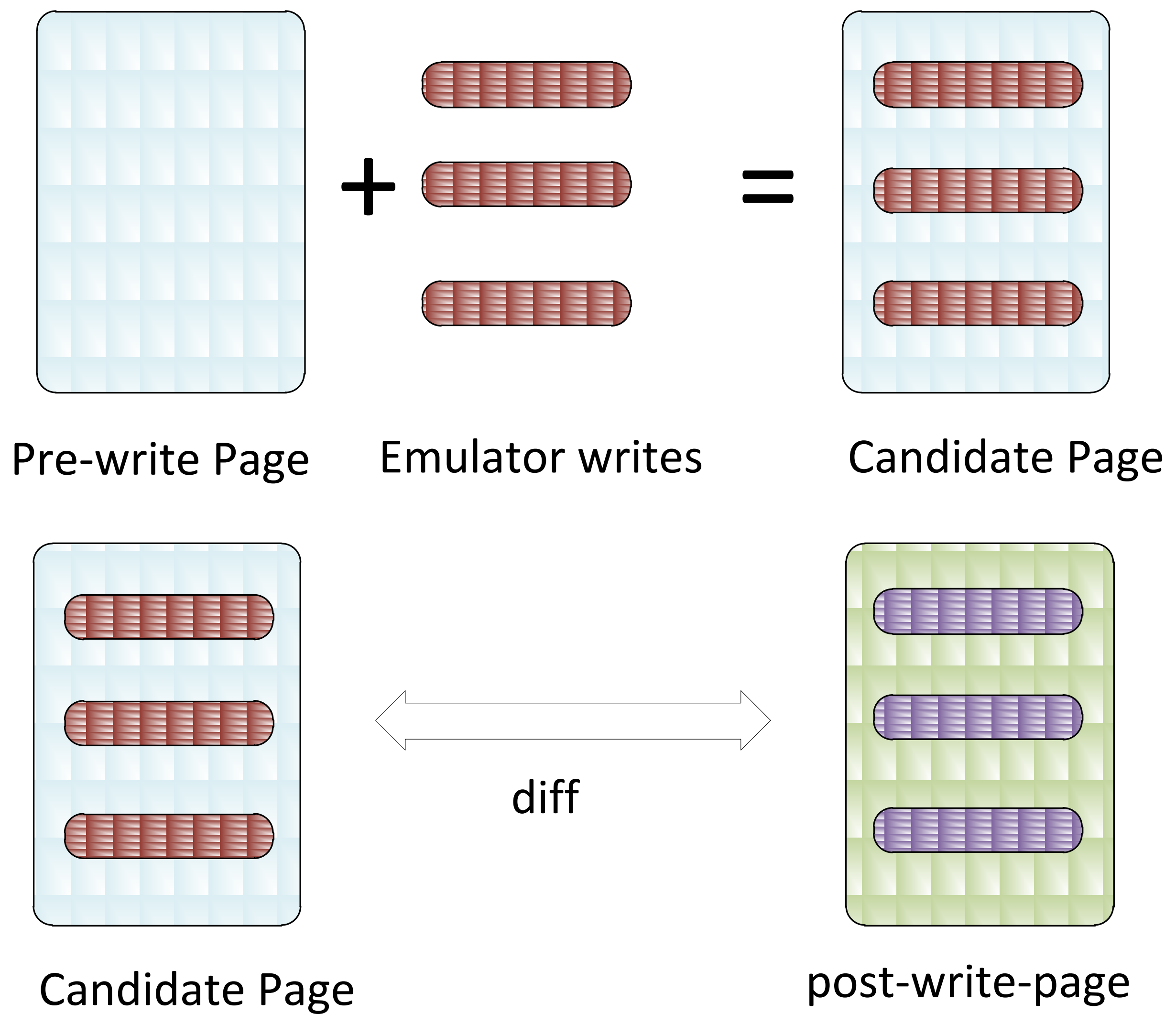
1. A debugger process (the validator)
2. A shared debugger helper library (libdebughelp.so)
3. Posix shared memory segments

10.2.3.1 Debugger Process

The debugger process uses "ptrace" and the /proc file system to control (and interrogate) the "test process" running on the physical CPU. It also instantiates a "TSL emulator", which it single steps in parallel with the debugged target process. The emulator utilizes a "look through semantics" for both register and memory state. That is, reads to unmodified registers and memory addresses are passed through to the underlying process, whereas writes are written to internal emulator variables. When testing an abstract interpretation the "alpha function" is applied on each "look through", thus providing the ability to efficiently model a process's entire 4 GB address space. To minimize the possibility of "cascading errors" the emulator is "re-synchronized" (the emulator state is flushed) to the physical CPU state after every single-step.

10.2.3.2 Memory Protection

Writes made by the emulator are detected by simply iterating over its internal data structures. Writes made by the target process are detected using a memory-protection scheme. The debugger write protects all pages in the test process, intercepts any segmentation faults, captures the pre-write values of any faulted pages, re-enables write permissions (if applicable), and then re-issues the faulting instruction. After each instruction executes, the modifications recorded by the emulator are then applied on top of the cached pre-written pages to produce "candidate pages". The "candidate pages" are then compared directly against the "post write pages" in the physical process. Any deltas imply the presence of a bug in the emulator. A similar technique is used for comparing register states.



10.2.3.3 Libdebughelp.so

The TSL validator is designed to support arbitrary programs, including programs that have no knowledge of the validator. Accomplishing that requires some care, however. The Linux System call interface does not provide any mechanism for a process to change the memory protections of an external process. Our goal for the validator is that it should "just work", and not require any kernel modifications. This implies that all code to enable and disable memory protection must be performed inside the target process via explicit calls to "mprotect".

In order to correctly validate a "concrete CPU semantics", the validator must detect ALL writes made to a process's memory, including its stack space. That introduces the additional burden that any code for manipulating a process's write protection must not perturb its stack space (so that it can be run when the stack is read-only). Unfortunately that precludes use of a normal functional call interface.

To get around this, the debugger uses a shared library (libdebughelp.so) to facilitate communication with the target process. It is dynamically injected into the test program using the

"LD_PRELOAD" environment variable. The Linux dynamic linker reads that variable and automatically "pre loads" any shared libraries it specifies into the process before transferring control to its entry point. The shared library includes an "elf constructor" that automatically maps a read-only shared page into the target process's address space (which is also mapped as read-write in the debugger process). The library also defines several hand-written assembly language routines that are carefully crafted to not perturb the target's stack. The debugger supplies arguments to those procedures by writing to the shared memory page, and transfers control to it using the "ptrace" api. Control is passed back to the debugger using the debugger interrupt ("int 3"). This provides a simple method for executing code in the target process without the complexities of dynamic code injection. It can also be used to run some operations (like write protection) "en masse", minimizing communication overhead (context switches).

10.2.4 Fuzz Testing

The TSL Validator also supports "fuzz testing". In fuzz mode, it supports specially crafted fuzz tests containing randomly generated instructions and supplies them with random inputs. Fuzz testing provides a simple automated means of generating interesting test cases for exercising CPU semantics. Unlike normal mode the goal of fuzz mode is not to preserve the exact semantics of a real program. Instead, the debugger tries, in earnest, to suppress all faults generated by the target.

Randomly generated programs, accessing random data, will read and write to random areas of a process's address space. Statistically, there is an extremely low likelihood that those addresses will be pre-mapped into the process[15]. In fuzz mode the debugger process compensates for this by intercepting those segmentation faults, generating pages of random data, and mapping them into the appropriate location in the target process.

Also, special care is needed to recover from execution of random instructions. It is generally desirable (for performance reasons) to execute multiple fuzz tests within a single process. However, random instructions that jump into random locations containing random data are inherently chaotic. To recover from this, the debugger must know the bounds of a given fuzz test. It needs to identify when the end of a fuzz test is reached, and it needs to know where the instruction pointer should be placed after a test to enable the next test to run. Furthermore any side effects of executing a fuzz test must be undone to prevent one test from corrupting others. Finally, random instructions may generate faults that would normally cause a process to terminate (like a divide by 0).

When run in fuzz mode, the validator handles those conditions. Fuzz tests employ a hand shaking protocol that communicates test scope and recovery information to the debugger. The

[15] Anecdotally, we observed about 28 4K pages mapped into the process for one small fuzz test. Given a uniformly distributed random 32 bit address, there is a probability of 99.997 % that any instruction accepting indirect operands would generate a segmentation fault.

debugger captures a process's state[16] before a test scope is entered, and restores it once a test completes. Finally, any faults generated within a test scope are suppressed by the debugger.

10.2.5 Tests

We ran several tests using this validator design, the results of which are shown in section 10.3 below. Our tests included 2 "real programs" ("Hello World", and a simple program that calls "exit(0)") and random fuzz tests for 10 categories of x86 instructions. We have excluded[17] from testing all non-deterministic instructions (such as those for accessing CPU counters), user mode instructions used only for interacting with an operating system, all privileged and real-mode instructions, all floating point instructions, all SIMD instructions, all "safer mode / VMX" extensions, and the "16 bit address size" forms of all instructions. The instruction categories presented in the results correspond, roughly, to the groupings used in the "Intel Opcode Map". There are also an additional 11 categories of instructions we do plan on testing that have not yet been implemented.

10.2.6 Limitations

There are a few limitations to our fuzz testing approach. By default, Linux does not make the entire virtual address space available for use in user mode processes. In particular, a typical 32 bit x86 Linux Kernel will reserve the lower 1 GB of virtual addresses for its own use. That, unfortunately, has the effect of decreasing our "signal to noise ratio (SNR)" by up to[18] 25%. To get around this we ran our tests as a 32-bit process on a 64-bit Linux system. For 32-bit processes on 64-bit Linux, the "reserved virtual address space" is only 64K, which for our purpose is effectively 0 (it results in an SNR degradation of about 0.002%). It is also possible to get similar results on a 32-bit system by using a specially configured kernel (with the HIGHMEM4G option enabled). We have not investigated the impact of these limitations on testing 64-bit processes, but they likely will require some changes to our test case generator.

[16] Memory changes are captured lazily, as updates occur.
[17] The exclusions were made because the corresponding functionality has not yet been implemented in our TSL specification.
[18] SNR: The ratio of tests for which we can measure results. The actual degradation (across our entire test suite) depends on the percentage of test cases that reference indirect operands (or are jumps) so in practice the real level of noise will be strictly less than 25%.

10.3 Experimental Results

Test	Total Instructions	Total Failures	Failure Rate	Unique Bug Count
exit(0)	1352	13	0.962%	2
"Hello world"	1337	75	5.610%	?
Nullary Fuzz-Tests	30	3	10.000%	8[19]
Group 1 Fuzz-Tests	176	5	2.841%	?
Group 2 Fuzz Tests	144	75	52.083%	26
Group 3 Fuzz Tests	63	18	28.571%	3
Group 4 & 5 Fuzz Tests	12	0	0.000%	0
Group 8 Fuzz Tests	40	10	25.000%	32[20]
Group 9 Fuzz Tests	4	4	100.000%	1
“Group P” Fuzz Tests	35	11	31.429%	?
“Group M” Fuzz Tests	99	11	11.111%	?

Table 1. Fuzz testing results

The table above summarizes the tests performed. We did not have time during the contract to determine the root cause all of the failures. In some cases a single problem manifested in many locations; in others a single poorly implemented semantic interpretation of an instruction had multiple problems. What is clear is that hand-generated semantic interpretations are as error-prone as any other computer-related endeavor – production of software, writing of tests, hand-review of source code, etc. Every artifact of this kind should have a mechanism for testing it and for maintaining its accuracy. With this proof of concept we now have a means to validate emulated interpretations as well.

10.4 Abstract Analysis

During the last month of the contract we focused on extending our test infrastructure to support abstract analyses in addition to concrete CPU semantics. This required several tasks:

1. Generalization of the test infrastructure.
2. Modifications to our analysis code to make it testable.
3. Instantiation of our “Simple VSA Analysis” within the validator.

Each task is described below.

[19] Some tests uncovered multiple bugs.
[20] Some tests uncovered multiple bugs.

10.5 Test Infrastructure

To support testing of abstract analysis we generalized our test infrastructure. This involved separating the “analysis neutral” and “analysis specific[21]” portions of our emulator loop, and factoring the analysis specific portions behind a common interface. The interface is shown in the figure below.

```
class IAnalysis
{
    friend class ref_ptr<IAnalysis>;
public:
    IAnalysis();
    virtual ~IAnalysis();
    virtual ref_ptr<IState> initialize(Debugger * pDebugger) = 0;
    virtual ref_ptr<IState> single_step(ref_ptr<IState> pState, Debugger * pDebugger) = 0;
    virtual void compare_state(ref_ptr<IState> pState, Debugger * pDebugger) = 0;
    virtual ref_ptr<IState> sync_with_cpu(ref_ptr<IState> pState) = 0;
protected:
    //count for ref_ptr
    unsigned int count;
};
```

10.6 Testability

To support testing of abstract analysis we had to make some modifications to improve their testability. TSL concrete CPU specifications represent registers, memory, and control flags using maps from memory addresses, register names, and cpu control flags to values. When a TSL specification is re-instantiated with an abstract domain, its concrete maps are replaced with abstract maps that also define additional operations needed during abstract interpretation. To support “look through” semantics for Simple VSA (Value Set Analysis), we had to add hooks (or fallback functors) to each of its abstract maps. Unlike the concrete case, this also required modifying their meet and join operators to properly handle partially specified maps[22].

10.7 Simple Value Set Analysis

Simple VSA is a simplified form of GrammaTech’s Value Set Analysis technology. It represents memory addresses and registers as a series of “strided intervals” (or “reduced interval congruencies”). A strided interval denotes a range of numbers [l,u] along with a “stride” s. For example, the strided interval ([0,6], 3) would include the numbers 0, 3, and 6 whereas the strided interval ([0, 10], 1) would include all of the integers from 0 to 10, inclusive. It also defines an abstract join operator that provides an over-approximation of set-union between two intervals,

[21] We define our support for “concrete CPU specifications” as a “concrete analysis”

[22] A map with a “functor” attached is “partially specified”. Some of its values are stored explicitly in a dictionary, while others are retrieved on demand from the functor. Conceptually, a map contains all the values that its fallback functor is capable of returning, even if those elements are not explicitly represented in the map. To maintain the mathematical properties necessary for sound analysis results, the implementations of “meet” and “join” on maps with “fallback functors” must take that conceptual view into account.

which is used to join intervals found on converging code paths. Finally, it also places a K-bound on the cardinality of the interval to ensure proper termination of the analysis[23]. This provides some of the functionality of "full VSA", while operating on substantially simpler abstractions.

10.8 Simple VSA Results

There were insufficient resources in the one-year project to complete the analysis of the results of the VSA validation. The validation was run on some of our test programs and did uncover additional bugs in the implementations of abstraction and representation functions.

[23] That is, if the result of joining 2 intervals would yield an interval that described a set with more than K elements, a value of "TOP" is returned instead.

11 Using the cloud to find malicious code

11.1 Introduction

"Cloud computing" has become a technical buzz-word in the last decade or so. It connotes an amorphous cloud of computing resources that can solve whatever computational need you may have. In fact, the concept of computing as a utility dates to the 1960's [88], a time when computers were expensive and computing resources were scarce. Time-sharing of computers was common-place, with users interacting only with input and output devices and not needing to own or manage or even know the location of the computer that did their work.

The advent of personal computers allowed individuals to have, cost-effectively, whatever computing power they needed under their own personal control and at their own convenience. However, most people use their computers for routine tasks – primarily email and document processing, though with increasing amounts of entertainment-oriented applications. Hence there is a great deal of unused computational power, left idle for the sake of convenience.

In enterprise computing, this unused computational power was seen as an opportunity for cost savings. By having "thin clients" at a business users' desks and the computation performed on centralized networks of servers, the cost of desktop units could be reduced and administration and maintenance of machines simplified.

The growth of high-bandwidth, always-connected internet connections across the country and the world has created the opportunity for new kinds of shared computing. Three in particular have gained prominence. First is the realization of computing as a utility [80]. Companies like Amazon and Microsoft provide access to computing services for a fee. Clients can rent time, memory and disk space as they use it; clients can rapidly be provisioned with additional facilities as they need them and ramp down when the need has passed. On an individual basis, the utility can provide applications for email and word-processing. Individuals need not even purchase software for their lightweight client machines – only an internet connection is needed. For large scale cluster computations, facilities like MPI can be included in the installation image and cluster programs can be run with little or no modification. In this type of shared computing, the computing resources used by individuals or business users are simply moved from their own desktops to a utility somewhere across the internet. But the amount or type of computation the users perform does not change.

In a similar, but non-commercial model, physically distributed grids of computational resources are made available by institutions such as the NSF- supported Terragrid and CERN's glite; these provide computing resources on a vast scale for tasks from all of the collaborators by utilizing dedicated resources at every institution [81-84]. Programs prepared for cluster computing can usually be transitioned by just relinking and producing a small set of scripts that describe the computational parameters.

A third type of shared computing results from making (usually donated) spare computational cycles on personal machines available to some larger project. In this case the project is typically a very large project with pieces of work that can be farmed out and the results collected back incrementally. Computing as a public utility comes with service guarantees and an infrastructure that guarantees privacy and security; computing with donated cycles has no such guarantees: resources may become available or unavailable without notice; security and privacy are not assured. However, research work has developed techniques to secure computations in a volunteer grid against accidental flaws and against intentional theft or sabotage [83]; the most popular cloud platform has built in facilities to support the devised methods [3, 7, 8, 86].

Mixed mode designs that use dedicated grids for some computations and volunteer computing for others have also been successfully deployed. Typically these involve a centralized job dispatch mechanism and some method for the user to tag subprocesses according to whether they are suitable for volunteer computing or not [85, 89].

Since the challenges associated with computing using donated cycles are much higher, including an estimated 3 work months of preparation time for new projects [3, 86], we will focus on this type of distributed computing in the remainder of this discussion. The application we have in mind is static analysis for finding malicious code.

11.2 Benefits of distributed computation

The benefits of highly distributed computation are well-known.

- A large problem can be tackled in a reasonable time, if it is sufficiently distributable.
- If the computational resources are donated, the cost is kept comparatively low [90, 91].
- Even if a public utility is used, the rate of progress and the amount of resources used can be varied.

Our application is the finding of bugs and malicious code within a large body of software using static analysis. Current static analysis tools use heuristics that miss bugs and report false positives in order to complete analyses in a reasonable time. Having orders of magnitude more computation available can increase the depth and precision of the analysis. We also see from our own studies in this project that it is possible to produce analysis results with low overheads and only infrequent communication. So, our application is, at least in principle, a good candidate for volunteer cloud computation.

There is also a potential intangible, social benefit to cloud computing. Volunteer participation in a larger project can encourage interest in the larger project; conversely, a well-positioned (or well-marketed) project can excite interest that leads people to volunteer resources. For example, participation in SETI@home or in research projects that study protein folding on a large computational scale or in the solution of large-scale mathematical problems is enhanced by the participants' interest in the social or scientific good of the project. In the right circumstances or with the right encouragement, the citizens of our country might be encouraged to participate in a large scale project that was important to the country's security – finding malicious code.

11.3 Requirements

However, not every task can be readily distributed to volunteer resources. The task must be able to be divided up into manageable subtasks and those subtasks must be able to be completed independently. There must be a high degree of intrinsic task parallelism [3].

Second, the overall computation must be robust against specific subtasks failing or being externally aborted. Tasks must not only be computed independently, but must be able to be completed in somewhat arbitrary order.

Third, the communication cost must be reasonably low. The task at hand must not require huge amounts of input data to be shipped to a volunteer computer, nor must the results consist of a large amount of data to be sent back to the controlling process. Large amounts of communication would be a bottleneck at the controller and would tax the volunteered computer. In addition it would diminish the value of distributed computing since the costs of communication would be larger compared to the costs of computation; ideally in parallel computation, communication costs are kept quite small.

Fourth, the subtasks must be tailored so as to not overly tax or monopolize the volunteered computer. In some situations, the computer on which the subtask is running may indeed be wholly devoted to the task; for example, someone may volunteer their computer for some pro bono computation at night, when the computer is not being otherwise used at all. However, in other circumstances, the donated cycles may be used for the distributed task concurrently with the user engaging in normal but not heavy use. In this case the externally supplied subtask must be content with running at a low priority, so as not to affect responsiveness to the rightful owner.

Finally, the external subtask must acknowledge and interact well with the security barriers on the local machine. After all, the external task is at one level no different than a bot that uses the local computer's resources for its own purposes, except that the bot is acting without permission. The security implications are discussed in more detail below.

Application to concolic execution

We observed in Section 7 that concolic execution is particularly amenable to distributed processing. In this application, a program executes a random input. The concolic engine observes a trace of that execution, notes which branches are taken during the execution, and then uses a constraint solver to determine what inputs would cause the execution to have taken other branch choices. The process is then repeated with these new inputs. The end result is a large test suite that provides high coverage of the subject program; it is particularly good at finding tests for rarely used (and potentially malicious) code branches.

The natural unit of work – the subtask – is the execution and analysis of a single input to produce other inputs. It requires the resources to execute a single run of the subject program and then to analyze that execution to produce new inputs. In fact those two steps could be separated and assigned to two different machines; the communication needed between them is just the trace of the execution.

All a subtask needs to begin work is the specific input to be analyzed. That includes the command-line arguments, any files that are read by the program, the data that flows to the program along any communication channels, including user input, and the sequence of events generated by a GUI. This is not typically a large amount of data. The response back to the controller is a similar set of data for each new input identified.

If a newly volunteered computer becomes available, all it needs to do is to accept work packets from the controlling process (along with installing the analysis program); if a volunteer suddenly becomes unavailable, with its work incomplete, the controller can simply restart that subtask with a new volunteer.

11.4 Technical and Social Challenges

The open nature of a system of computing using donated cycles requires a number of security issues and operational pitfalls to be addressed. In this section we discuss the technical and social challenges to effective use of cloud computing for malicious code discovery.

As mentioned above, the DoD Defense Science Board Task Force issued a report on "Mission Impact of Foreign Influence on DoD Software." The summary findings of the report present a stark view of the critical importance and incredible difficulty of malware detection. From a theoretical point of view, we believe the question, "does this software have a *behavior* that indicates *malicious intent*?" is actually impossible to answer. Any (interesting) question about software *behavior* quickly runs afoul of Rice's Theorem and is undecidable. *Malice* is often dependent on context. For example, a program that broadcasts one's current GPS coordinates might be a cool new app for the iPhone; the same behavior on a weapon system is almost certainly treasonous. Finally, simply defining *intent* is likely to lead to a philosophical morass.

However, more importantly, we agree with the Defense Science Board that mitigating steps can and should be taken. In particular, we believe that recent breakthroughs in machine-code analysis will prove to have strategic importance. For a long time, machine-code analysis was effectively limited to *dynamic* techniques that work by observing a program as it executes. Dynamic techniques can give precise information for the observed executions — but *only* for the observed executions. Recently, meaningful *static* analysis of machine code has become viable. Static analysis performs an abstract execution to determine properties that hold for all possible runs of a program. In regards to machine-code analysis, there are novel static-analysis techniques for recovering an *Intermediate Representation* (IR) that captures a program's semantics [13, 14, 59]. The recovered IRs have a plethora of uses [12, 16], including (but not limited to) checking for behaviors that indicate malicious intent [1, 52].

We believe that the next breakthrough in machine-code analysis can be found by leveraging the power of online communities. Dynamic and hybrid dynamic-static analyses can benefit by multiplying the number of executions that are available for analysis. Static analysis can benefit by multiplying the computational power that is available to the analysis. However, building an effective computational community presents its own set of challenges. The remainder of this section summarizes some of these challenges.

11.4.1 Algorithmic Requirements

The first challenge is to adapt the analysis algorithms for distributed computation (across the community). For many analyses, this is straightforward. For example, for some dynamic algorithms, it may be as simple as partitioning a program's input space and assigning each community member to analyze the program when run on a different portion of the input space.

On the opposite end of the spectrum, many analysis problems are thought to be difficult to parallelize efficiently. For example, it is believed that (unless $P=NC$) there are limitations on creating algorithms for interprocedural dataflow analysis that are (a) parallel, (b) efficient (in utility of the parallel computations), and (c) precise [76]. However, even for these analyses, we should not lose hope of benefiting from a computational community. For example, a community might allow for a sequential algorithm to be applied to more individual problems. Each community member might perform the entire (sequential) analysis of a single program, but together the community is able to analyze the entire suite of programs used on a given platform.

Furthermore, precise program analysis is often theoretically impossible but this has not stopped the development of useful analyses. Usually, the analysis succeeds by sacrificing precision. Similarly, by using abstractions and sacrificing (perfect) precision, it may be possible to develop static analyses that are parallel, efficient, and usually precise in practice.

One reason for optimism is the success of demand-driven analyses [39, 51, 73, 74]. A demand-driven analysis computes answers only at specific points of interest in a program. For example, rather than computing the possible points-to relations for all variables in a program, a demand-driven points-to analysis might focus on computing only the points-to relations for the variables used in the innermost loops (that are targeted for the heaviest optimization). In theory, a demand-driven analysis may have to compute the complete solution just to get a point solution. However, in some cases demand-driven analyses — even when executed for every conceivable demand — can be faster than traditional exhaustive analyses [23]. (This is because an exhaustive analysis may waste resources propagating information where it cannot be used.) In other cases, demand analyses are not as competitive [74]. Nevertheless, the results of [23] suggest that one possible approach to leveraging computational communities is to assign each community member to run a demand-driven algorithm on a different set of (related) demands, and then assemble the individual solutions into a complete solution.

11.4.2 Security Concerns

Cloud computing raises security concerns on a number of levels: safety of the individual community members, integrity of distributed computations, protection from espionage, and trustworthiness of the borrower.

11.4.2.1 Safety of Community Members

As in the Hippocratic Oath, the first maxim of any community-based approach should be "do no harm." Participation in an (online, computational) community should be safe. At a minimum, this requires guarantees that:

- The stability of a participant's system is not affected.
- The participant is not exposed to new attack vectors.
- The participant controls the cost of participation. (I.e., the participant sets the allocation of her resources in the community effort.)
- The participant's privacy is never compromised.

In many cases, these guarantees are easy to provide. For example, if participants are called upon to run partial computations, then those computations should be "sandboxed," or run in a virtual environment that isolates the computation. Proper sandboxing can ensure that a program cannot make unauthorized access to system resources and that any fallout from running a faulty program is contained to the sandbox. (There are many effective techniques for sandboxing a program, such as using a hypervisor or using *process shepherding* [53], e.g., using GrammaTech's *software dynamic translator* [11], gtSDT.)

Sandboxing helps to maintain the stability of the participant's system. It may also be important for ensuring that participants are not exposed to new attack vectors. One possible approach to identifying malicious code is to use dynamic analysis (such as "smart fuzzing") to attempt to trigger a malicious behavior. This technique becomes much more powerful if an entire computational community is running the dynamic analysis — but it comes with an obvious risk. If the analysis succeeds (malicious code is triggered), then one or more community members will be running malicious code on their systems. By running the dynamic analysis (and any malicious code) in a sandbox, we can guarantee that no harm is done.

In some cases, providing the guarantees listed above will require additional research. In particular, it is difficult to guarantee privacy when gathering feedback from community members. In general, the feedback gathered from participants would be related to the execution of (specific subject) programs on the participants' machines. Just the fact that a participant is using a particular program with a particular frequency may be sensitive information. Worse, facts about a program's execution may imply facts about sensitive data that the program was examining or manipulating. This problem is magnified if the program was running on classified data; any feedback about an execution on classified data may have to share the same classification.

In some cases, concerns about the loss of privacy can be mitigated by using an anonymizing Tor network to submit feedback data. However, anonymous submission of feedback does not help if the feedback data is classified or somehow identifies the submitter. For some feedback-based techniques, it may be possible to prove that the feedback data does not reveal any sensitive information. The *Cooperative Bug Isolation* project samples feedback data in order to decrease the flow of information about each execution and lessen the risk of privacy leaks [60].

In the end, concerns about safety (such as the leaking of private information) may restrict participation in a community (e.g., by members of the intelligence community). We expect community-based approaches that restrict participation to still be beneficial — even to those members who do not participate. (Solutions found by community members benefit everyone.)

11.4.2.2 Masquerading testers

Besides being protected from harm, a person donating cycles must trust that the organization using the donated cycles is well-behaved and honest in its intentions. Given the difficulty of knowing who is whom on the internet, such knowledge and trust is a challenge. A computational (or any) community is built on the social fabric that the result of donated resources is something worthwhile. The donor must be able to obtain confidence that such is the case and have transparency into the results of the research.

11.4.2.3 Integrity of Distributed Computations

In addition to providing protection *to* its members, a community-based effort may also require protection *from* its members. For example, a malicious community volunteer might try to subvert a community-wide computation by submitting false results, or slow progress by requesting work assignments but never fulfilling them.

Fortunately, the integrity of a distributed computation can often be protected with simple techniques [83], including:

- *Replication of work assignments*. Each work assignment should be randomly assigned and replicated among multiple community members. This makes it much more difficult for a malicious participant to submit false results without being detected.
- *Use of CAPTCHAs to screen membership in a community.* CAPTCHAs are a common technique across the web for distinguishing 'bots' from humans. Community-based

approaches should leverage CAPTCHAs, just as online services do. For example, use of CAPTCHAs can prevent a botnet from disrupting a community-based effort.

- *Random audits of member work*. Another way to identify falsified results is to randomly check a small percentage of results by reproducing the result on a trusted machine.
- *Consensus of feedback*. Feedback-based approaches might benefit by not trusting any datum until it has been submitted by multiple community members. As with replication of work assignments, this helps to protect against false reports.

The above approaches are fairly generic across all community-based techniques. For specific community-based analyses, it might be possible to take additional precautions. For example, it is often cheaper to check if a result is correct than it is to compute the result from scratch.

11.4.2.4 Protection from Espionage

Public research often faces the danger that the research will be used for nefarious purposes. This has often been the case in computer science. *Fuzz testing* was originally conceived as a technique for improving software development [4, 65]. Today, “fuzzing” is a popular technique among hackers for finding exploitable security vulnerabilities.

Community-based analyses face a similar threat that they could be appropriated and misused. In particular, community-based efforts could be subject to a form of “espionage” where a “spy” joins the community to gain access to the community’s software. The spy could then create his own community that competes with the original community (e.g., to find malicious backdoors or software vulnerabilities). If the spy uses botnets to create his community, then he might gain a computational advantage over a community that relies solely on volunteers. Another danger of espionage is that by examining the analysis algorithm, the spy might be able to construct malicious code that will escape detection.

The espionage can also take the form of a man-in-the-middle attack. The spy might not directly participate in the community effort, but simply observe the traffic that occurs, obtaining information about what tasks are being performed, what the results of analyses are, and even who is participating in those tasks.

When possible, steps should be taken to minimize susceptibility to espionage. For example, in many distributed computations, there is a centralized component that (a) distributes work parcels and (b) assembles distributed results. The centralized component is much harder to steal. If the algorithm is designed so that the centralized controller is also hard to implement or to reverse-engineer, then the threat of espionage is greatly reduced.

11.4.3 Community Creation and Coherence

As described above, not all of the hurdles in this project involve shortcomings in technology. Some of the challenges are also social or political. In particular, community-based analyses are effective in direct proportion to the size of the participating community. This means that building and maintaining a large community of volunteers is important to the success of any community-based analysis.

Fortunately, there are some reasons to be optimistic that this challenge is surmountable. There already exist online communities that have similar participation requirements (although we are not aware of any that address the problems we are investigating). The BOINC project [3] at UC Berkeley and the World Community Grid [5] from IBM both provide a centralized location where volunteers can join computational communities of their choice. This means that by developing our analyses with the appropriate APIs, we will be able to leverage existing efforts to recruit and organize computational communities. Furthermore, we believe that computer security is a problem that affects everyone and we should be able to find volunteers. Still, there will be questions that need to be addressed. For example, which programs get analyzed? With what priority? Ensuring that these questions are answered fairly may be important for maintaining community coherence.

Internet denizens are also becoming more used to participating by providing feedback about their program behavior. Larus reports that the Dr. Watson tool receives feedback from approximately 42% of users [55]. However, it is still important to make it easy for volunteers to sign up and participate. For example, all that Dr. Watson requires of a participant is a few mouse clicks.

11.5 Conclusions

There's an old adage in computer science that "if brute force doesn't solve your problems, then you aren't using enough." For the problem of identifying malicious code, there have been promising techniques — static model checking, concolic execution, etc. — but they have all suffered from scalability problems. They did not have enough "brute force" to solve realistic problems. One of the easiest ways to leverage a community is to use the spare computational cycles of the community members: the community provides the necessary brute force.

There are many advantages to approaches that build and leverage computational communities. In particular, it is relatively easy to guarantee the safety of community members, especially with regards to privacy issues. Similarly, it is relatively easy to ensure the integrity of the distributed computation. On the other hand, computational communities may be vulnerable to espionage, as described in Section 11.4.2.4.

Computing in the cloud or with donated spare cycles can offer significant computing resources to a problem such as finding useful test cases using concolic execution. Concolic execution is admirably suited to using distributed resources. However, the security concerns in doing so may limit the use of such widely distributed processing to studying programs that are publicly available anyway, not those in which there is high proprietary or classified value. The same distributed technique can also be used on a captive set of computer servers, with the same advantages and more control over the available computational cycles and over security.

Section III: Concluding Material

12 Related Research

This section describes related work in three areas. We begin with a discussion of related work in the areas of static and dynamic analysis. Next, we describe work on mining temporal specifications. Finally, we discuss some implementations of the paradigms of passive feedback and donation of spare computation cycles.

12.1 Static and Dynamic Analysis

Both static and dynamic approaches have emerged to address the problem of detecting security vulnerabilities in software. Static approaches include ITS4 from Cigital (formerly Reliable Software Technologies) [77]. ITS4 uses surface-level syntactic information to detect constructs that can lead to a wide range of potential problems including buffer overruns and file-based race conditions. The tool does not attempt to do any static-semantic analysis beyond a coarse approximation to the call graph. RATS and Flawfinder use similar techniques to ITS4. Another static approach is that of Wagner et al. [78].

Dynamic approaches focus on detecting attacks in progress and preventing them from doing harm. StackGuard [36], and a similar technology in Microsoft's C/C++ compiler for Visual Studio .NET prevent buffer-overflow attacks from succeeding by detecting or preventing damage to the stack at runtime. The same research group that developed StackGuard, also developed RaceGuard [35], which is a kernel-level patch that monitors system calls and aborts the program when it detects that a race-condition attack is in progress. Immunix's FormatGuard [34] provides a safe standard library to compile against.

Another dynamic approach used for detection of anomalous or malicious code has been intrusion detection systems (IDSs) [94]. An “intrusion” is detected by observing key behavior of an application, like the system calls made by it. Any deviation from the expected would then be raised as an intrusion.

Approaches from academia that have analyzed source code for (presumably unintentional) malicious code are Wagner's BOON (which inspired GrammaTech's buffer overrun detector) and Engler's ARCHER for detecting buffer overrun vulnerabilities, RacerX [41] and Dilger's tool for finding race conditions [21], Chen's MOPS for finding security problems like misuse of the 'seteuid' function, and extended static checking [27] for finding a range of security problems.

Detection of intentionally malicious code in source or binary code has been tackled via data mining [71] and pattern matching via static analysis [29].

12.2 Mining Temporal Specifications

In this section, we give examples of two approaches to mining temporal specifications, such as interface specifications. The first observes dynamic traces of program execution, while the second performs static analysis of the subject program.

12.2.1 Mining Dynamic Information

Originally these techniques were used to extract models from traces of function calls for the purposes of documentation or maintenance, as described by Cook and Wolf [31], El-Ramly et al. [40], Mariani et al. [63], and Ernst [44]. More recently, work has been done on mining API call traces to derive specifications that describe how the API should be used correctly. This specification is then used as input to a program verifier. Ammons et al. describe the technique [6], and Yang and Evans extend the approach to include any temporal property [79]. This approach has been extended further to cover mining of other dynamic program properties. Bowring et al. describe its use on execution branches and paths [22].

12.2.2 Mining Static Information

One of the first applications of machine learning for the purpose of finding software flaws is due to Engler et al. [43]. In their approach, a system learns simple static properties that hold most of the time, and alerts the user to exceptions. This is based on the observation that many bugs are correlated with anomalies. For example, if a program does things one way 99% of the time, yet slightly differently in the remaining 1%, then the outliers are anomalies and are statistically likely to contain flaws. Properties learned by the system fit templates such as "*function **a** must be paired with function **b***". This system has been successful at finding large numbers of flaws in operating-system software; it has been transitioned to the Coverity product line.

12.3 Passive Feedback for Cooperative Bug Isolation (CBI)

The CBI project [57, 58] aims to improve the quality of software by means of the passive-feedback paradigm. The project distributes versions of a few popular open-source applications that were instrumented to collect a variety of predicates (or events) that could help in isolating the cause of a crash. An example predicate is the information that the return value of a function is negative. Each execution of the instrumented program reports (sampled) counts of the number of times that each predicate was true during the execution, as well as the exit status of the execution. CBI uses statistical machine-learning techniques to infer likely causes of a crash. Intuitively, the algorithm looks for sets of predicates that co-exist in failing runs but not in successful ones. These sets of predicates are likely indicators of crash-inducing bugs.

The CBI project is faced with many of the same types of issues that we expect to encounter, namely concerns of privacy, performance, and adoption. Heavy use of sampling alleviates the performance overhead, as well as privacy concerns to a large degree.

As mentioned above, a significant challenge to widespread use of CBI is due to the need for source code during the instrumentation step. Currently, only a small collection of CBI-instrumented open-source software packages are available, and only for selected Linux distributions. Additionally, the use of instrumented binaries makes the presence of CBI detectable by intruders: a savvy intruder may elude CBI by choosing a different (non-instrumented) target.

12.4 Efforts to Harness Donated Computation Cycles

BOINC [3, 7, 8] is a software system that enables scientists to tap into the vast computing resources of personal computers. Individuals can donate their computers' resources to one ore more BOINC projects, with the ability to specify the allocation of the donated resources among these projects. BOINC originated as an infrastructure to support SETI@home (Search for Extra-Terrestrial Intelligence) but was later extended to become a generic middleware platform for volunteer and grid computing. BOINC hosts a number of projects, primarily in the areas of mathematics, medicine, molecular biology, climatology, and astrophysics.

World Community Grid (WCG) [5] is an effort by IBM Corporation to create the world's largest public computing grid. It grew out of a massively distributed computing grid that was used to accelerate the search for a cure for smallpox. The success of the smallpox effort led IBM to launch WCG as a computational framework to benefit a variety of humanitarian research efforts with high computational requirements. Initially, WCG only supported Windows and relied on a proprietary distributed computing package (Grid MP from United Devices). To open access to computational resources that use other platforms, WCG added the support of the BOINC interface, thus enabling users of computers running Mac OS X and Linux to volunteer cycles.

We are not aware of any program-analysis projects that build upon the BOINC framework. While that framework may be an appropriate foundation for our work as well, some particularities of our intended applications may warrant a new effort on the foundation of the framework. A common theme among many of the current BOINC projects is the fact that they address problems whose solutions are not likely to benefit any adversary. For instance, it is hard to imagine a person who can benefit from compromising (or hijacking) the computation of the N-queens mathematical problem or the search for a cure for smallpox. In the case of program analysis aimed at identifying malice in software, it is clear (as discussed earlier) that an adversary can benefit by: (a) preventing an analysis from functioning correctly (e.g., when hiding a known exploit), and (b) stealing the computed results (e.g., when finding new exploits).

13 Conclusions and Recommendations

The Safety In Numbers project conducted research related to several aspects of using donated, distributed resources to assess the quality of software and to find malicious code. The principal conclusions are enumerated here:

- Using donated resources is a feasible approach to garnering a large computation resource. However it does have some social and security challenges. The principal technical challenge is that the task to be performed must be sufficiently parallelizable.
- We assessed whether conventional static analysis could make efficient use of a wide network of computational capability. We concluded that there is some benefit to applying more time to an analysis than users typically are willing to wait, but that the benefit is only modest. To obtain significant benefit from a parallel architecture would require at minimum a substantial re-architecting of the logical representation of the subject program.
- By contrast, concolic execution of a subject program in order to find a test suite that gives a high degree of execution path coverage (and is thus able to find hidden, possibly malicious, execution paths) is very amenable to distributed and volunteer execution. We found concolic execution (as implemented in GrammaTech's GraCE tool) to be reasonably scalable with relatively low communication costs. The parallel efficiency is high and is expected to be higher with larger scale programs. Details of the search algorithms for finding novel execution paths had unexpected implications for the efficiency of various parallel execution architectures.
- Finally, we demonstrated that we could adapt the techniques of concolic execution and random "fuzz" testing to validate logical representations of the semantics of concrete systems. This is an essential component of developing confidence that the static analysis tools are operating correctly. We demonstrated this random fuzzing for two problems. First, applying concolic execution to a machine code program requires a logical representation of the machine code; our validator was able to check the translation of a significant subset of the hundreds of x86 instructions, finding bugs in the logic that have been referred for inspection. Second, we demonstrated a proof of concept that the same technique can be applied to a more abstract interpretation of the machine semantics as well.

The research conducted in this project was challenging and provocative; it sparked many more ideas than could be followed up in the ten-month span of work. Some of those ideas are listed here.

- The exercise of validating the concrete semantics of machine code instructions demonstrated the validation technique, but the work remains of understanding the discrepancies discovered and folding that information into a validated machine semantics. The result will be a thoroughly checked semantics of x86 that can be used in a number of tools.

- Similarly the assessment of the abstract interpretation through fuzzing techniques was a successful proof of concept that now can be applied to a wider range of abstract interpretations.
- The experiments in parallelizing concolic execution successfully demonstrated that the tool could be applied to a moderate scale set of computing resources. No roadblocks in the architecture are apparent from our testing. There are two steps needed to apply this technique at a realistic scale. First to further extend the capability of the GraCE engine, and second to validate the predictions made in this project by applying the architecture developed here in systems containing hundreds of nodes.
- Concolic execution has shown its promise admirably in a number of research groups. Consequently it is worth supporting its development past the proof of concept to full-fledged commercial tool. We will be seeking further contract support to commercialize this promising technology in a way that encourages widespread use for analyzing software binaries.
- Our original 3-year proposal proposed using user communities to obtain feedback on bugs in applications. Such cooperative mechanisms are already used for crash reports (e.g. Dr. Watson) and for usage reports (e.g. Eclipse); one could also apply this technique to advantage for test cases and static analysis results.

14 References

1. Christodorescu,M., Jha,S., Maughan,D., Song,D.X., and Wang,C., eds. *Malware Detection*. Advances in Information Security, ed. Jajodia,S. Vol. 27. 2007, Springer: New York City, NY. 312.
2. *Mission Impact of Foreign Influence on DoD Software*, 2007, The Defense Science Board.
3. BOINC: Open-Source Software for Volunteer Computing and Grid Computing, http://boinc.berkeley.edu/.
4. Software Testing with Holodeck using Fault Injection, http://www.securityinnovation.com/holodeck/index.shtml.
5. World Community Grid: Technology Solving Problems, http://www.worldcommunitygrid.org/index.jsp.
6. Ammons,G., Bodík,R., and Larus,J.R., *Mining Specifications.* In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2002. Portland, OR: ACM. pp. 4-16.
7. Anderson,D.P., *BOINC: A System for Public-Resource Computing and Storage.* In *Workshop on Grid Computing*. 2004. Pittsburg, USA.
8. Anderson,D.P. and Fedak,G., *The Computational and Storage Potential of Volunteer Computing.* In *IEEE/ACM International Symposium on Cluster Computing and the Grid*. 2006. Singapore.
9. Anderson,P., Reps,T., and Teitelbaum,T., *Design and Implementation of a Fine-Grained Software Inspection Tool*. IEEE Transactions on Software Engineering (TSE), 2003. **29**(8): pp. 721-733.
10. Anonymous, The Folding@Home Project, http://folding.stanford.edu/.
11. Bala,V., Duesterwald,E., and Banerjia,S., *Dynamo: A Transparent Dynamic Optimization System.* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2000. Vancouver, British Columbia, Canada: ACM. pp. 1-12.
12. Balakrishnan,G., Gruian,R., Reps,T., and Teitelbaum,T., *CodeSurfer/x86 -- A Platform for Analyzing x86 Executables.* In *International Conference on Compiler Construction (CC)*. 2005. Edinburgh, UK: Springer. pp. 250-254.
13. Balakrishnan,G. and Reps,T., *Analyzing Memory Accesses in x86 Executables.* In *International Conference on Compiler Construction (CC)*. 2004. Barcelona, Spain: Springer Verlag. pp. 5-23.
14. Balakrishnan,G. and Reps,T., *DIVINE: DIscovering Variables IN Executables.* In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2007. Nice, France: Springer. pp. 1-28.
15. Balakrishnan,G. and Reps,T., *Analyzing Stripped Device-Driver Executables.* In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008. Budapest, Hungary: Springer. pp. 124-140.
16. Balakrishnan,G., Reps,T., Kidd,N., Lal,A., Lim,J., Melski,D., Gruian,R., Yong,S.H., Chen,C.-H., and Teitelbaum,T., *Model Checking x86 Executables with CodeSurfer/x86 and WPDS++, (tool-demonstration paper).* In *International Conference on Computer Aided Verification (CAV)*. 2005. Edinburgh, Scotland: Springer. pp. 158-163.
17. Balakrishnan,G., Reps,T., Melski,D., and Teitelbaum,T., *WYSINWYX: What You See Is Not What You eXecute.* In *IFIP Working Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. 2005. Zurich, Switzerland: Springer.
18. Ball,T. and Rajamani,S.K., *The SLAM Toolkit.* In *International Conference on Computer Aided Verification (CAV)*. 2001. Paris, France: Springer Verlag. pp. 260-264.
19. Ball,T. and Rajamani,S.K., The Slam Project: Debugging System Software via Static Analysis, http://research.microsoft.com/slam/papers/popl02.pdf.
20. Beckman,N.E., Nori,A.V., Rajamani,S.K., and Simmons,R.J., *Proofs from Tests.* In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2008. Seattle, WA: ACM. pp. 3-14.
21. Bishop,M. and Dilger,M., *Checking for Race Conditions in File Accesses*. Computing Systems, 1996. **2**(2): pp. 131-152.
22. Bowring,J.F., Harrold,M.J., and Rehg,J.M., *Improving the Classification of Software Behaviors using Ensembles of ControlFlow and DataFlow Classifiers*. 2005, College of Computing, Georgia Institute of Technology, Atlanta, GA GIT-CERCS-05-10.
23. Bruening,D., Duesterwald,E., and Amarasinghe,S., *Design and Implementation of a Dynamic Optimization Framework for Windows.* In *ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*. 2001. Austin, TX.

24. Bush,W.R., Pincus,J.D., and Sielaff,D.J., *A Static Analyzer for Finding Dynamic Programming Errors*. Software - Practice and Experience (SPE), 2000. **30**(7): pp. 775-802.
25. Chen,H., Dean,D., and Wagner,D., *Model Checking One Million Lines of C Code.* In *Symposium on Network and Distributed System Security (NDSS)*. 2004. San Diego, CA: The Internet Society. pp. 171-185.
26. Chen,H. and Wagner,D., *MOPS: An Infrastructure for Examining Security Properties of Software.* In *ACM Conference on Computer and Communications Security (CCS)*. 2002. Washington, DC: ACM. pp. 235-244.
27. Chess,B., *Improving Computer Security Using Extended Static Checking.* In *IEEE Symposium on Security and Privacy*. 2002. Oakland, CA: IEEE Computer Society. p. 160.
28. Choi,J.-D., Lee,K., Loginov,A., O'Callahan,R., Sarkar,V., and Sridharan,M., *Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs.* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2002.
29. Christodorescu,M. and Jha,S., *Static Analysis of Executables to Detect Malicious Patterns.* In *USENIX Security Symposium*. 2003. Washington, DC: USENIX Association. pp. 169-186.
30. Clarke,E.M., Fujita,M., Rajan,S.P., Reps,T., Shankar,S., and Teitelbaum,T., *Program Slicing for Design Automation: An Automatic Technique for Speeding-up Hardware Design, Simulation, Testing, and Verification.* In *Conference on Correct Hardware Design and Verification Methods (CHARME)*. 1999. Bad Herrenalb, Germany: Springer. pp. 298-312.
31. Cook,J.E. and Wolf,A.L., *Discovering Models of Software Processes from Event-Based Data*. ACM Transactions on Software Engineering and Methodology (TOSEM), 1998. **7**(3): pp. 215-249.
32. Corbett,J.C., Dwyer,M.B., Hatcliff,J., Pasareanu,C.S., Robby, Laubach,S., and Zheng,H., *Bandera : Extracting Finite-state Models from Java Source Code.* In *International Conference on Software Engineering (ICSE)*. 2000. Limerick, Ireland: ACM. pp. 439-448.
33. Cousot,P. and Cousot,R., *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints.* In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1977. Los Angeles, CA: ACM. pp. 238-252.
34. Cowan,C., Barringer,M., Beattie,S., and Kroah-Hartman,G., *FormatGuard: Automatic Protection From printf Format String Vulnerabilities.* In *USENIX Security Symposium*. 2001. Washington, DC: USENIX Association. pp. 191-200.
35. Cowan,C., Beattie,S., Wright,C., and Kroah-Hartman,G., *RaceGuard: Kernel Protection from Temporary File Race Vulnerabilities.* In *USENIX Security Symposium*. 2001. Washington, DC: USENIX Association. pp. 165-176.
36. Cowan,C., Pu,C., Maier,D., Hinton,H., Walpole,J., Bakke,P., Beattie,S., Grier,A., Wagle,P., and Zhang,Q., *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.* In *USENIX Security Symposium*. 1998. San Antonio, TX: USENIX Association. pp. 63-78.
37. Daily, M., *IBM, Harvard Want Your PC for Solar Power Study*. 8 A.D.
38. Das,M., Lerner,S., and Seigle,M., *ESP: Path-Sensitive Program Verification in Polynomial Time.* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2002. Berlin, Germany: ACM. pp. 57-68.
39. Duesterwald,E., Grupta,R., and Soffa,M.L., *Demand-Driven Computation of Interprocedural Data Flow.* In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1995. San Francisco, CA: ACM Press. pp. 37-48.
40. El-Ramly,M., Stroulia,E., and Sorenson,P., *From Run-Time Behavior to Usage Scenarios: An Interaction-Pattern Mining Approach.* In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2002. Edmonton, Alberta, Canada: ACM Press. pp. 315-324.
41. Engler,D.R. and Ashcraft,K., *RacerX: Effective, Static Detection of Race Conditions and Deadlocks.* In *ACM Symposium on Operating Systems Principles (SOSP)*. 2003. Bolton Landing, NY: ACM Press. pp. 237-252.
42. Engler,D.R., Chelf,B., Chou,A., and Hallem,S., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions.* In *Symposium on Operating System Design and Implementation (OSDI)*. 2000. San Diego, CA: USENIX Association. pp. 1-16.

43. Engler,D.R., Chen,D.Y., and Chou,A., *Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code.* In *ACM Symposium on Operating Systems Principles (SOSP)*. 2001. Banff, Alberta, Canada: ACM. pp. 57-72.
44. Ernst,M.D., Cockrell,J., Griswold,W.G., and Notkin,D., *Dynamically Discovering Likely Program Invariants to Support Program Evolution.* In *International Conference on Software Engineering (ICSE)*. 1999. Los Angeles, CA: ACM. pp. 213-224.
45. Festa,P., *RSA: 56-bit Crypto Too Weak.* in *CNET News*. October 23, 1997.
46. Ganapathy,V., Jha,S., Chandler,D., Melski,D., and Vitek,D., *Buffer Overrun Detection using Linear Programming and Static Analysis.* In *ACM Conference on Computer and Communications Security (CCS)*. 2003. Washington, DC: ACM Press. pp. 345-354.
47. Godefroid,P., Klarlund,N., and Sen,K., *DART: Directed Automated Random Testing.* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2005. Chicago, IL: ACM. pp. 213-223.
48. Godefroid,P., Levin,M.Y., and Molnar,D., *Automated Whitebox Fuzz Testing*. 2007, Microsoft MSR-TR-2007-58.
49. Havelund,K. and Pressburger,T., *Model Checking Java Programs Using Java PathFinder*. International Journal on Software Tools for Technology Transfer (STTT), 2000. **2**(4): pp. 366-381.
50. Henzinger,T.A., Jhala,R., Majumdar,R., and Sutre,G., *Lazy Abstraction.* In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2002. Portland, OR: ACM. pp. 58-70.
51. Horwitz,S., Reps,T., and Sagiv,M., *Demand Interprocedural DataFlow Analysis.* In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 1995. Washington, DC: ACM. pp. 104-115.
52. Kinder,J., Ktzenbeisser,S., Schallhart,C., and Veith,H., *Detecting Malicious Code by Model Checking.* In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2005. Vienna, Austria: Springer. pp. 174-187.
53. Kiriansky,V., Bruening,D., and Amarasinghe,S., *Secure Execution Via Program Shepherding.* In *USENIX Security Symposium*. 2002. San Francisco, CA: USENIX. pp. 191-206.
54. Lal,A., Reps,T., and Balakrishnan,G., *Extended Weighted Pushdown Systems.* In *International Conference on Computer Aided Verification (CAV)*. 2005. Edinburgh, Scotland, UK: Springer. pp. 434-448.
55. Larus,J.R., *The Real Value of Testing.* In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2008.
56. Liblit,B., *Cooperative Bug Isolation: Winning Thesis of the 2005 ACM Doctoral Dissertation Competition*. Lecture Notes in Computer Science. Vol. 4440. 2007: Springer.
57. Liblit,B., Aiken,A., Zheng,A.X., and Jordan,M.I., *Bug Isolation via Remote Program Sampling.* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
58. Liblit,B., Naik,M., Zheng,A.X., Aiken,A., and Jordan,M.I., *Scalable Statistical Bug Isolation.* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
59. Lim,J. and Reps,T., *A System For Generating Static Analyzers for Machine Instructions.* In *International Conference on Compiler Construction (CC)*. 2008. Budapest, Hungary: Springer. pp. 36-52.
60. Lim,J., Reps,T., and Liblit,B., *Extracting Output Formats From Executables.* In *Working Conference on Reverse Engineering (WCRE)*. 2006. Benevento, Italy: IEEE Computer Society. pp. 167-178.
61. Loginov,A., Yahav,E., Chandra,S., Fink,S.J., Rinetzky,N., and Nanda,M.G., *Verifying derefence safety via expanding-scope analysis.* In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2008.
62. Loginov,A., Yong,S.H., Horwitz,S., and Reps,T., *Debugging via Run-Time Type Checking.* In *International Conference on Fundamental Approaches to Software Engineering (FASE)*. 2001. Genova, Italy: Springer. pp. 217-232.
63. Mariani,L. and Pezzè,M., *Inference of Component Protocols by the kBehavior Algorithm*. 2004, Università degli Studi di Milano - Bicocca. Dipartimento di Informatica, Sistemistica e Comunicazione. Laboratorio di Test e Analisi del Software, Milano. Technical Report LTA:2004:05.
64. Melski,D. and Reps,T., *Interconvertibility of a Class of Set Constraints and Context-Free Language Reachability*. Theoretical Computer Science, 2000. **248**(1-2): pp. 29-98.

65. Miller,B.P., Fredriksen,L., and So,B., *An Empirical Study of the Reliability of UNIX Utilities*. Communications of the ACM, 1990. **32**(12): pp. 32-44.
66. Miller,B.P., Koski,D., Lee,C.P., Maganty,V., Murthy,R., Natarajan,A., and Steidl,J., *Fuzz Revisited: A Re-Examination of the Reliability of UNIX Utilities and Services*. 1995, University of Wisconsin-Madison, Madison, WI.
67. Ramalingam,G., Field,J., and Tip,F., *Aggregate Structure Identification and Its Application to Program Analysis.* In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1999. San Antonio, TX: ACM Press. pp. 119-132.
68. Reps,T., Balakrishnan,G., and Lim,J., *Intermediate-Representation Recovery from Low-Level Code.* In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. 2006. Charleston, SC: ACM Press. pp. 100-111.
69. Reps,T., Balakrishnan,G., Lim,J., and Teitelbaum,T., *A Next-Generation Platform for Analyzing Executables*. in *Malware Detection*, Christodorescu,M., Jha,S., Maughan,D., Song,D.X., and Wang,C., Editors. 2007, Springer: New York. pp. 43-61.
70. Reps,T., Schwoon,S., Jha,S., and Melski,D., *Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis*. Science of Computer Programming, 2005. **58**(1-2): pp. 206-263.
71. Schultz,M.G., Eskin,E., Zadok,E., and Stolfo,S.J., *Data Mining Methods for Detection of New Malicious Executables.* In *IEEE Symposium on Security and Privacy*. 2001. Oakland, CA: IEEE Computer Society. pp. 38-49.
72. Sen,K., Marinov,D., and Agha,G., *CUTE: A Concolic Unit Testing Engine for C.* In *European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. 2005. Lisbon, Portugal: ACM. pp. 263-272.
73. Sridharan,M. and Bodik,R., *Refinement-Based Context-Sensitive Points-To Analysis for Java.* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
74. Sridharan,M., Gopan,D., Shan,L., and Bodik,R., *Demand-Drive Points-To Analysis for Java.* In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
75. Teitelbaum,T. and Reps,T., *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment*. Communications of the ACM, 1981. **24**(9): pp. 563-573.
76. Thomas Reps, *On the sequential nature of interprocedural program-analysis problems*. Acta Informatica, 1996. **33**: pp. 739-757.
77. Viega,J., Bloch,J.T., Kohno,T., and McGraw,G., *ITS4: A Static Vulnerability Scanner for C and C++ Code.* In *Annual Computer Security Applications Conference (ACSAC)*. 2000. New Orleans, LA: IEEE Computer Society. p. 257.
78. Wagner,D., Foster,J.S., Brewer,E.A., and Aiken,A., *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.* In *Symposium on Network and Distributed System Security (NDSS)*. 2000. San Diego, CA: The Internet Society. pp. 3-17.
79. Yang,J. and Evans,D., *Dynamically Inferring Temporal Properties.* In *ACM SIGPLAN-SIGSSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 2004. Washington, DC: ACM. pp. 23-28.
80. Doe, J.: Cloud Computing and Software Services (2009) http://netlib.org/utk/people/JackDongarra/PAPERS/GridSolve_chapter.pdf
81. Grandinetti (ed), Lucio. Grid Computing: The New Frontier of High Performance Computing. Elsevier Science and Technology Books, Inc.. © 2005. Books24x7. <http://common.books24x7.com/book/id_17807/book.asp> (accessed November 15, 2010)
82. Abbas, Ahmar. Grid Computing: A Practical Guide to Technology and Applications. Cengage Charles River Media. © 2004. Books24x7. <http://common.books24x7.com/book/id_7274/book.asp> (accessed November 15, 2010)
83. Silaghi, G., Araujo, F., Silva, L., Domingues, P., Arenas, A.: Defeating Colluding Nodes in Desktop Grid Computing Platforms. Parallel and Distributed Processing, 2008. IPDPS 2008.
84. Sottrup, C., Pederson, J.: Developing Distributed Computing Solutions Combining Grid Computing and Public Computing
85. Lattice Homepage. http://boinc.umiacs.umd.edu/

86. Meyers, D., Bazinet, A., Cummings, M.: Expanding the Reach of Grid Computing: Combining Globus- and BOINC-Based Systems (2007)
87. McConnell, Steve. *Code Complete, 2nd Edition.* Redmond, Wa.: Microsoft Press, 2004.
88. Parkhill, Douglas. *The Challenge of the Computer Utility.* Addison-Wesley, 1960.
89. Bazinet, A.: The Lattice Project: A Multi-Model Grid Computing System (2009). University of Maryland Masters Thesis
90. Kondo, D., Javadi, B., Malecot, P., Cappello, F., Anderson, D.: Cost-Benefit Analysis of Cloud Computing versus Desktop Grids. IPDPS '09 Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing
91. Andrzejak, A., Kondo, D., Anderson, D.: Exploiting Non-Dedicated Resources for Cloud Computing.
92. Majumdar, R., K. Sen. Hybrid Concolic Testing. ICSE, 2007.
93. Thakur, A., et al. "Directed Proof Generation for Machine Code". CAV, 2010.
94. Hoffmeyr, S. A., S. Forrest, A. Somayaji. Intrustion Detection Using Sequences of System Calls. J. Computer Security, 1998.